DATA REPRESENTATION AND SYNTHESIS

Fred M. Tonge and Lawrence A. Rowe

Technical Report #63

March 1975

Department of Information and Computer Science
University of California, Irvine

The substance of this technical report is taken from a research proposal submitted to the National Science Foundation. The authors are particularly indebted to Professor Thomas A. Standish who participated in many of the discussions leading to this proposal, contributing both ideas and specific prose.

ABSTRACT

This research concerns the problem of specifying the information relationships and their transformations included, explicitly and implicitly, in any problem-solving procedure. Our view of data representation is that problem representations (in a problem domain) are mapped to a machine representation (in an implementation domain) through various modelling representations (in modelling domains) Modelling domain representations make easier the discovery of acceptable implementation representations. We propose to focus on the study of mappings of data representations and their transformations from the modelling to implementation domains. Specifically, our goal is to answer four questions:

1) what are appropriate formalisms for describing modelling domains and implementation domains,
2) what knowledge exists of each of these domains and how can this knowledge be represented in our formalisms,
3) how do we map specific representations from one domain to the other, and
4) how can we test the completeness of our formalisms and mapping process.

In answering the fourth question, we propose to construct an interactive system for generating alternative implementation domain representations from a modelling domain representation and selecting that which comes closest to the user's desired program performance criteria in an actual programming context.

INTRODUCTION AND OVERVIEW

Purpose of this Research

The research described in this proposal concerns representing a problem domain for computer solution. The general problem is that of specifying to a machine a procedure to solve a problem, i.e. the programming problem. The specific part of this problem of interest here is that of representing the information relationships (data) and their transformations which are present, explicitly and implicitly, in the problem specification and in the solution algorithm.

The programming problem, and particularly the subproblem of data representation, will not be solved in a short time by any single research group. Rather it requires a continual effort by a number of people, each contributing to and building upon other's results. Accordingly, we wish to choose a focus for our research which will have leverage in the sense that its findings can be used by others to advance the state of the art of programming. This focus is aimed at deepening our understanding of data representations. Ultimately, we wish to understand data representations with sufficient clarity and precision that, for a significant range of programming problems, we can describe the choices of data representations available for

their solution and select those representations that yield the best quantitative performance properties for a given intended application.

Our View of Data Representation

Problem Domains

Programming problems arise naturally in a diverse collection of problem domains. Some problem domains, such as orbital mechanics required for space shots, are understood with sufficient scientific precision that the formal theories required for writing programs are abundant. By contrast, some other problem domains, such as medical diagnosis, are not precisely understood in sufficient detail to enable us to write computer programs that we could substitute trustfully for specialists. Between these two extremes lie a spectrum of problem domains with varying degrees of formalization and theory.

Implementation Domains

Given that problem solution procedures must ultimately be reduced to data and instructions executable by computers, the implementation medium chosen provides a domain into which solutions must eventually be reduced. Implementation structures may arise at the machine level or at the programming language level, depending on the choice of

implementation medium. At whatever level chosen, implementation structures constitute the ultimate target of reductive transformations which represent algorithms as executable programs.

Modelling Domains

Often in the process of devising solutions to programming problems posed in problem domain terms, use is made of intermediate structures. Such intermediate structures serve as media capable of expressing algorithms, but typically are only indirectly executable by a computer. We call such intermediary domains "modelling domains."

For example, to solve the problem of charging for a multi-site teletype network, with teletypes placed in distinct cities, a telephone company might use the concept of a "minimal spanning tree." A computer program that computes the charges might represent such a minimal spanning tree internally using "nodes" and "pointers." In this case, "teletypes" and "lines" are concepts at the problem domain level, "minimal spanning tree" is a concept at the modelling domain level, and "nodes" and "pointers" are concepts at the implementation domain level.

Thus, the programming process involves a cascade of reductions from problem domain concepts into implementation structures perhaps utilizing one or more layers of modelling

structures.

Problem Domains	-->	Modelling Domains	-->	Implementation Domains

Representations

Each time we map a structure at one of these levels into a structure at a lower level, we invoke the notion of a "representation." A representation is a use of a set of objects and processes at one level to imitate relevant behaviors of objects and processes at a higher level. The notion of representation is nicely captured through the notions of "interpretations" and "models" in contemporary mathematical logic [18]. In this research we are particularly interested in the representations of problem data.

Some Specific Questions

We propose to focus our efforts on the study of mappings of representations from modelling domains to implementation domains. We see here a number of avenues that are ripe for exploration and likely to yield significant progress.

More specifically, the goals of our proposed research are to find answers to the following questions:

* What are appropriate formalisms for describing modelling domains and implementation domains?

* What knowledge exists of each of these domains and how can this knowledge be represented in our formalisms?

* How do we map specific representations from one domain to the other?

* How can we test the completeness of our formalisms and mapping process?

Summary of Proposed Research

We have already made progress on the first of the above questions, that of formally describing modelling domains and implementation domains. We are pursuing an approach to describing modelling domains which consists of a language for expressing structural constraints and required relationships between elements in an abstract structure together with a set of primitive operators for expressing transformations of these structures [22]. The work on this "modelling structure" formalism has been completed. As described below, we must now assess its usefulness in different applications.

We propose to develop a parallel formalism for describing implementation domain representations.

With regard to the second of the above questions, what knowledge do we have of the modelling and implementation

domains, we have begun to construct a taxonomy of modelling structures using our modelling formalism [22]. We are able to characterize such diverse modelling structures as lists, graphs, trees, sets and strings in terms of distinct characteristic features described by the formalism. This will permit us to catalog and index a wide range of structures in the modelling domain.

A goal of the proposed research is to build a catalog of various representations both at the modelling level and at the implementation level.

For each structure described at the abstract (modelling) level, there are typically several reasonable implementation structures that can be used to represent them accurately at a more concrete (implementation) level. For example, Schwartz has described various ways of implementing sets [23]. Numerous papers in the literature describe alternative methods for representing lists [13, 16, 29]; and, Knuth characterizes various types of trees, such as free trees, oriented trees, ordered trees, binary trees and extended binary trees [15]. At the very least, the descriptive adequacy of our formalism must be measured against these examples. Furthermore, we must demonstrate how our formalisms can capture and describe significant differences at both the modelling and implementation levels.

Given success in devising an adequate descriptive formalism, we must then attempt to answer the third of the above questions, how can we map from modelling domains into implementation domains? At the very least we can collate the already considerable body of knowledge of representation techniques into a relational structure. This cross-index or catalog would relate specific modelling structures to a range of possible implementation structures. But this alone is not enough. Because we can not hope to catalog all possible structures at either the modelling or the implementation level, (and indeed such a goal would be unreasonable), we must also devise an algorithm for generating alternative implementation structures for a given modelling structure. And because the choice among such alternatives must be made rationally, we must be able to evaluate alternatives with respect to their requirements for space and time.

This leads to the last question listed above, can we test the completeness of our achieved understanding? To test our achieved understanding, we propose to develop a system which, in concert with the user, generates alternative implementation level representations and selects a representation meeting the user's desired program performance criteria. Our work on this system aims to

illustrate two points: first, the interactive selection and synthesis of implementation level representations, involving a dialog with knowledgeable users, and second, the capability of feeding back to the user measured program performance properties for his use in refining the choice of modelling and implementation structures.

More detailed descriptions of these questions and of our proposed approach are given in the technical discussion section below.

Expected Significance of the Proposed Research

Today, and for the immediate future, hardware costs seem to be declining rapidly (by a factor of 10 every 5.7 years according to Roberts [21]), whereas software costs are an increasingly dominant fraction of the cost of new systems. Currently, the software/hardware cost ratios for many new systems run from 3:1 to 7:1, and Boehm estimates that in the absence of remedial measures, by 1985 these ratios are likely to approach 9:1 [3]. In fact, every year since 1966 the Federal Government has spent more on software than on hardware [20], and software production is at least a 10 billion dollar a year industry (over one percent of the GNP). Thus, the evolution of an effective science of programming has important economic consequences.

It is widely recognized that we need to find improved techniques for producing and maintaining reliable software, especially since software requirements seem to become progressively more ambitious and since software production is, at present, highly labor intensive. While short-term improvements in software productivity might well be brought about by better management techniques [3], software production would still be labor-intensive. In the long run, significant improvements in programmer productivity will rest upon making the computer a more talented partner in the process of program production.

An important ingredient in achieving this partnership is to develop ways of programming the computer to fill in most low-level programming detail by itself, allowing much less detailed specifications of procedures than are required at present. If the description of programs is made more compact by the omission of detail that either programmers or machines can fill in, then programming tasks of all sorts are made easier. This is partly because there is less low-level mechanical detail to obstruct understanding and creation in such tasks as composing, reading, editing, maintaining and debugging programs.

The research we are proposing is aimed at enabling low-level representation details to be handled either

automatically or interactively with much less detail than possible at present. Thus, it should be possible for programmers to say less to specify a particular product, and so, for a given amount of effort, to achieve more.

TECHNICAL DISCUSSION

In this section we discuss five specific problem areas in more detail, including related results to date, other work in progress, our goals, and steps to achieve these goals. The first two areas are the modelling structures and implementation structures formalisms as approaches to describing modelling and implementation domains. The taxonomy/catalog as a representation of knowledge about modelling and implementation domains is the third. The fourth problem area is the mapping of modelling structures to a range of possible implementation structures. The fifth is the development of an interactive system to generate implementation structures from specified modelling structures in an actual programming context.

Formalizing Modelling Domain Knowledge

The proposed approach to formalizing modelling domain knowledge is to describe a set of modelling structures and operations for transforming such structures one into another. We use the term modelling structure to describe a structure which models the salient aspects of the information from a problem domain used by an algorithm to solve a problem. Because it is a modelling level concept, a modelling structure is independent of any commitments to the

implementation of the structure. In this approach we describe a class of abstract objects and abstract operations having a range of possible underlying concrete representations. The formalism provides a convenient mechanism for selecting subsets of the set of modelling structures, the desired abstract objects, and a small set of primitive operators for defining the desired abstract operations. In a recent report [22] (attached as an appendix), we describe the modelling structures formalism, including a rigorous definition of the primitive operators. While a first version of the formalism is complete from the standpoint of the research being proposed here, experience in using it will undoubtedly suggest improvements. In the following section we informally summarize this descriptive framework.

Modelling Structures Formalism

A modelling structure is either a primitive object or a structured object. A primitive object is an instance of a data type provided as primitive by the problem domain. Thus, in one domain square arrays may be primitive; whereas, in another domain they are structured objects. Conventional programming languages provide primitive data types such as integers, floating points, booleans, and characters.

A modelling structure which is a structured object is

composed of elements and relations among the elements. The elements of a modelling structure are instances of modelling structures. Thus, structured objects may be composed of primitive and structured objects.

There are six properties used to describe modelling structures. These are:

1 replication of element values in a structure,

2 ordering of elements in a structure,

3 structural relations between elements,

4 distinguished elements in a structure,

5 referencing of elements in a structure, and

6 operations applied to a structure and its elements.

A structured object may allow replication of elements and may be ordered (an order predicate defines an ordering between the elements).

Relations between elements describe how the structure is put together. Associated with each relation is a set of attributes, each attribute representing a piece of information associated with the relation between two specific elements in a structure. The number of relations defined on a structure and the number of attributes for a particular value of a given relation are not limited. For example, to model a road map composed of cities and roads between cities, we might define a modelling structure in

which the elements represent the cities and a relation, ROAD, represents the roads. Associated with each road would be several attributes describing that particular road, such as the distance between the cities, the highway number or numbers, and the speed limit. This data is represented by attributes associated with the ROAD relation.

Associated with each structured object are two means of referencing elements in the structure, external accesses and distinguished elements. External accesses are references to particular elements in a structure. Distinguished elements are element references based on the relations defined for the structure. They are bound to elements by way of the structure and thus may change as the structure changes. In contrast, regardless of how a structure changes, an external access references the same element as long as it remains in the structure unless explicitly changed. Distinguished elements are useful for representing such structural conepts as the head of a list. External accesses are useful for moving through a structure from element to element along relation links.

The primitive operators for transforming one instance of a modelling structure into another include: insert an object into a structure (insert), delete an object from a structure (delete), replace one object in a structure by

another (replace), create an external access to an object in a structure (createaccess), relate two objects by a given relation (relate, includes insert if either object is not a member of the structure), unrelate objects within a structure (unrelate), return one or all objects related to a particular object by a specified relation (related), read an attribute of a relation (readattr), and store an attribute of a relation (storeattr).

To illustrate this formalism, we will define a set, a 1-way list, and a stack.

A set does not allow replication nor may an ordering be defined on its elements (otherwise it would be an ordered set). There are no relations or distinguished elements. Elements in the structure are referenced by quantification (essentially universal, "for all ...," and existential, "there exists ...," quantifiers). The operations allowed on a set are read, insert, delete, and replace.

A 1-way list is different from a set in that replication is allowed, there is a defined relation, and two distinguished elements. The relation is SUC, which is 1-1, has a single element not in the domain of SUC, has a single element not in the range of SUC, and is connected (all elements may be reached by SUC or its implicit inverse). The two distingished elements are an element not in the

range of SUC, "head," and an element not in the domain of SUC, "tail." Furthermore, the forms of referencing and operations allowed are different than for a set. Elements are referenced by external accesses and distinguished elements rather than by quantification. The operations allowed are read, delete, replace, createaccess, relate, and related.

The third example is a stack. A stack is identical to a 1-way list except for the way elements are referenced (limited to the distinguished element "head") and the permitted operations (constrained to read, delete, and a limited form of relate). These examples show how this formalism can be used to provide a taxonomical framework for modelling structures.

There has been a significant amount of work on data representation in the last few years. The work by D'Imperio [7, 8], Kapps [14], Mealy [17] and Turski [27] is directed towards a theory of data structures emphasizing the meaning of data and the fundamentals of structures and computation. Many of the concepts represented in these models are used in our modelling structures formalism. Bobrow and Raphael [2], Earley [10], Shneiderman and Scheuermann [25], Taft and Standish [26], van Wijngaarden [31], Wegbreit [30], and Wirth [33] are representative of the extensive work on data

definitional facilities and structure operations in programming languages.

Formalizing Implementation Domain Knowledge

A formalism for expressing possible implementations of modelling level representations must include the notions of indivisible cells (capable of representing primitive values) and of groupings of cells into structures. Such structures may be implemented using contiguous groupings of cells (referenced by a conventional indexing mechanism), cells which point explicitly to other cells or groups of cells (referenced by a conventional indirect mechanism), and cells whose value serves as an argument to a structure-defining function (referenced by a hashing mechanism). Because at the machine level these cells are not unconnected entities, they must also be treated as belonging to an ordered memory.

There are two types of implementation domain knowledge we need to represent: 1) alternative implementation structures for particular modelling structures, and 2) methods of organizing several implementation structures into a coherent whole (e.g. a region of ordered memory).

As an example of the first type, a stack may be represented either using contiguous cells to represent elements in the stack and an index to the top of the stack, or using non-contiguous groups of two cells to represent

elements in the stack, each group including the stack element and a pointer to the group representing the previous element in the stack, together with a pointer to the group representing the top element in the stack. Examples of knowledge about representing structures in a region of memory are described in Knuth [15].

The formalism for representing alternative implementations is relatively straight forward. In fact, conventional data definitional facilities in extensible programming languages, such as PPL or ECL, may be adequate. An adequate descriptive formalism for knowledge about organizing implementation structures into coherent wholes is not so obvious. In this research we plan to collect examples of such knowledge and incorporate it into a system for generating implementation structures from modelling structures (see below). Whether this knowledge is to be incorporated as "expert" procedures or whether it can be distilled into a general process operating on a detailed knowledge base is an open question at present.

Representing Modelling and Implementation Domain Knowledge

One phase of this research program is to collect and organize the diverse body of knowledge on modelling and implementation structures available in the computer science literature. This knowledge is a part of the information

base which the proposed implemention structure generation system uses. The formalism developed for modelling structures and that to be developed for implementation structures will serve as the basis for organizing this knowledge. A start on describing modelling structures in this formalism has been made [22].

We do not believe that this taxonomy/catalog will be precise, in the sense that all users will agree with its descriptions and classifications. (Note the current variety of definitions of such concepts as "stack" or "list".) Rather, we anticipate a situation similar to that in biology where the overall focus and major divisions of plants and animals are agreed on, but particular plants or animals are classified in different ways by different people. As in biology, the importance is not the particular taxonomy/catalog, the importance is how well we can use it.

Mapping Modelling Level to Implementation Level Representations

A major problem area is that of describing a number of possible implementation structures for a given modelling structure. This can be viewed as mapping modelling domain knowledge to implementation domain knowledge. For those modelling structures that are classified in the taxonomy/catalog, specific alternative implementation level

representations as described in the implementation structures formalism can be associated in the catalog. For those modelling structures not so classified, an alternative implementation structure generator is needed. Space and processing time costs (probably relative figures of merit) associated with different implementation structures can also be stored in the catalog or generated for use in alternative evaluation, as discussed in the following section.

Results to date on this problem of relating modelling to implementation domain knowledge include the wealth of alternative implementation structures used in different programming language translators [11, 12, 27, 32] and which have appeared in the literature [4, 6, 15, 19]. Recent work by Shneiderman [24] can possibly be adapted as the alternative implementation structure generator.

Generating and Selecting Implementation Level Representations for Modelling Level Representations

To test the representational approach described above, we propose to construct a system which, operating in concert with the user, can produce an implementation level representation for a given program which meets the desired performance criteria. If successful, such a system could improve both the programmer's productivity and the quality of programs produced.

Difficulties with Current Practice

Currently programs are written partly at the modelling level (the program control structure and some operations) and partly at the implementation level (the data structures and related operations). This occurs because the formalisms available for expressing information relationships and data in programming languages actually represent specific implementation structures. Therefore a programmer must select implementation structures at the time he produces a program level representation.

Forcing the programmer to choose an implementation structure at this stage of the programming process causes several problems. The first problem has to do with the clarity of the program. Most modelling structures have several natural underlying implementation structures. These "natural" implementations are often easy to understand. However, the mapping of a modelling structure by a particular language translator may result in an inefficient implementation. In fact, the natural implementation structure might not even be supported by the language. Consequently, the modelling structure cannot be described directly in the language, but rather must be "twisted," possibly obscuring the intent of the program. Because modelling structures, and hence implementation structures,

appear partly in the data and partly in the program, a change in implementation structure may require substantial change in the program.

A second problem concerns the time at which implementation structures are chosen. Since the choice is usually made before any coding of the program, the programmer may not clearly understand all the implications of the different possibilities and he may overlook some important detail. Also at this stage the programmer is often interested only in getting a correct representation. Thus, he may ignore or postpone decisions concerned with efficiency.

Another problem concerns debugging and proving the program correct. If a modelling structure must be represented in terms of lower level primitives, then the modelling structures and operations on them must be reprogrammed and proved correct with each usage. It is true that this difficulty can be somewhat alleviated through the use of libraries of common routines; however, it is often the case that the library routines are not widely known, are not well documented, or do not quite provide the facility that the programmer desires.

Finally, programmers often do not spend the time and effort necessary to carry out analyses of different

implementation structures. Frequently, it is not until after the program is running and fails to meet efficiency criteria that an analysis is conducted. This is much too late.

Computer Assisted Selection of Implementation Structures

Our proposal to solve some of these problems is to provide representation of information relationships and data at the modelling structure level in a program, and then to use the computer to assist in searching for acceptable implementation structure representations. This solution embodies the philosophy that programmers should write their procedures in a machine independent formalism and, where possible, should not explicitly represent low level implementation decisions. Under this approach, the problem, not the permissible language structures, determines the appropriate modelling structures for a program.

The second part of this solution, using the computer to assist in selecting efficient implementation structures, involves both ill-defined objects (the different problem solution procedure representations at the modelling and implementation levels) and an ill-defined objective function for selecting among the objects. When faced with this type of ill-structured problem there are two possible approaches.

The first is to define constraints on the objects and the objective function so that the problem is no longer ill-defined. The second approach is to define some partial constraints and allow the user to interact with the system to guide it in the search for a solution. The first approach is appropriate when a reasonable set of constraints can be established which allows the system to solve a meaningful subset of the overall problem. The second approach is appropriate when a reasonable set of constraints, or formalism, cannot be found. Because of the difficulty of defining acceptable constraints, we feel a solution along the lines of the second approach has the greater potential. The approach is a mixture of a man-machine interaction which allows cooperation between the two parties in order to improve the problem solving process.

A system for generating and selecting among alternative implementation structures is yet another tool in a programming environment. A good programming environment includes a wide variety of facilities, namely, editors, translators, debugging tools, documentation aids and measurement facilities for the development, testing, and maintenance of programs. In the proposed system, programs are written and debugged in an interpretive environment, using modelling structure level representations. After a

program has been debugged, it can be compiled to produce a more efficient implementation. The (temporary) implementation structures used in the interpretive mode for the different modelling structures are not particularly important. At this stage the user is concerned with developing a correct program and thus is more interested in minimizing elapsed time for the translation than in efficiency of the execution. The result of this program writing phase is a complete program and modelling domain representation for the problem solution. The second phase, or compilation, takes this representation as input and produces an implementation domain representation for the problem solution based on the efficiency criteria given by the user. The two phases are called, respectively, the program writing and implementation-selection phases.

Writing the Program. In the program writing phase the user enters statements to the system. The statements can be program statements, direct commands interrogating or changing the system environment, or a request to enter the implementation-selection phase. Statements describing the properties of different modelling structures are direct commands which change the environment of the system. Information is maintained on the modelling structures used by the program as statements are entered. The system

anticipates structure and tries to complete the modelling structure definitions in terms of known (cataloged) structures. It also tests the consistency of structure definitions and accessing. Thus the user may specify his structure definitions before writing his program, as he writes it, or when he has finished writing it. Definitions can be examined and changed at any time. Examples of properties of a structure being defined that the system tries to fill in are whether the structure is ordered, the types of objects in the structure, and the maximum size of the structure.

The modelling structures formalism, as described above, allows users to define a structure either as some known modelling structure maintained in a catalog or to create a structure by defining the properties it must have. Examples of modelling structures in the catalog are arrays, stacks, queues, and lists. Properties by which a structure may be defined are the size of the structure, the manner in which it is referenced, ordering of the structure, types of objects in the structure, and whether elements in the structure may be repeated. Those users who know which specific modelling structure they want may specify it by name. Those users who are not aware of the full catalog of specific modelling structures, or who do not know exactly

which modelling structure they want, may define the properties of their modelling structure for the system. In either case the system is continually checking the consistency of and gathering information about the structures.

Generating and Selecting Implementation Structures. The implementation-selection phase generates the implementation structure representation of the program. The system generates a set of alternative implementation structures for each distinct instance of a modelling structure used in the program. Then an evaluation algorithm selects one implementation structure for each modelling structure and the program is compiled using these implementation structures and appropriate accessing routines.

Alternative implementation structures are generated by first matching each modelling structure used in the program with those defined in the catalog of known structures. The catalog includes a complete definition for each modelling structure and a set of alternative implementations. For those modelling structures which do not match a structure in the catalog, a general alternative generation algorithm is used. The catalog of modelling structures is used to embody detailed knowledge about possible implementations, since

knowing what a specific modelling structure is may result in a more efficient implementation structure than would be provided by the general alternative generator. Thus a major component in the system is the algorithm for deducing what modelling structure is being used. This matching is trivial when a modelling structure used in a program is named, for example stack, list, array, set or queue. The difficult case is when the structure is described by properties that must hold for the structure (e.g. relationships between elements in the structure and how the structure and its elements are referenced) and by the manner in which the structure is operated upon.

The alternative selection component gathers information needed to analyze the possible implementation sets. (An "implementation set" is a set containing a choice of one implementation structure for each modelling structure.) Sources for this information are modelling structure definitions and declarations, static flow analysis of the program, sample runs and interactive interrogation of the user. The expected costs, both in space and time, for each alternative implementation set is calculated and that which comes closest to meeting the desired efficiency is selected. If the estimated cost is not acceptable to the user, a set of heuristics is used for suggesting changes in the

modelling structure definitions which might lead to less costly implementations.

Often the user may not have approximate values for some inputs to the estimated cost function. In these cases the user may choose to have the system provide data gathering routines in some test runs to allow the user to perform an experiment to estimate the parameters. The system inserts the appropriate data gathering routines, and the user then executes the program supplying sample test data. Examples of the types of data which might be gathered are the frequencies of insertions, deletions, searches and changes to a structure.

Other Approaches

There have been other proposals to build this type of system, but to our knowledge none has been completed [1, 5, 9, 10]. Our approach to this goal factors a large ill-structured problem into several smaller, more structured problems which are more amenable to formalism and analysis. At the same time, while we intend to pursue the introduction of formalism and analysis as vigorously as possible, we do so in the context of allowing and indeed soliciting interaction with the problem-solver.

REFERENCES

1 Balzer, R. Dataless Programming. Proc. AFIPS FJCC, vol. 31 (1967), pp. 535-544.

2 Bobrow, D. G. and B. Raphael. New Programming Languages for Artificial Intelligence Research. Computing Surveys, vol. 6, no. 3 (September 1974), pp. 155-174.

3 Boehm, B. W. Software and Its Impact: A Quantitative Assessment. Datamation, vol. 19, no. 5 (May 1973), pp. 48-59.

4 Brooker, R. A., D. Morris and J. S. Rohl. Trees and Routines. Computer Journal, vol. 5, no. 4 (April 1962), pp. 33-47.

5 Cheatham, T. The Recent Evolution of Programming Languages. Proc. IFIP Congress (1971), pp. I.118-I.134.

6 Deuel, P. On a Storage Mapping Function for Data Structures. Comm. of the ACM, vol. 9, no. 5 (May 1966), pp. 344-347.

7 D'Imperio, M. E. Data Structures and their Representation in Storage. In Annual Review in Automatic Programming 5, Pergamon Press, New York, NY (1969), pp. 1-75.

8 D'Imperio, M. E. Information Structures: Tools in Problem Solving. Unpublished paper (July 1969).

9 Earley, J. Towards an Understanding of Data Structures. Comm. of the ACM, vol. 14, no. 10 (October 1971), pp. 617-627.

10 Earley, J. Relational Level Data Structures for Programming Languages. ACTA Informatica, vol. 2 (1973), pp. 293-309.

11 Gries, D. Compiler Construction for Digital Computers. Wiley, New York, NY (1971).

12 Hopgood, F. R. A. Compiling Techniques. American Elsevier Publishing Co. Inc., New York, NY (1969).

13 Iverson, K. E. A Programming Language. Wiley, New York, NY (1962).

14 Kapps, C. A. SPRINT A Programming Language with General Structure. PhD Thesis Moore School Report No. 71-18, The Moore School of Electrical Engineering, Univ. of Penn. (August 1970), pp. 94-123.

15 Knuth D. E. The Art of Computer Programming, Vol. 1. Addison-Wesley, Reading, MA (1968).

16 McCarthy, J. et. al. Lisp Programmer's Manual. MIT, Cambridge, MA (August 1962).

17 Mealy, G. M. Another Look at Data. Proc. AFIPS FJCC, vol. 31 (1967), pp. 525-534.

18 Mendelson, E. Introduction to Mathematical Logic. Van Nostrand, Princeton, NJ (1964).

19 Morris, R. Scatter Storage Techniques. Comm. of the ACM, vol. 11, no. 1 (January 1968), pp. 38-44.

20 Pierce, J. R. Computers in Higher Education. Report of the President's Science Advisory Committee (February 1967).

21 Roberts, L. G. Personal communication.

22 Rowe, L. A. Modelling Structures Formalism. Technical Report #52, Dept. of Information and Computer Science, U. C. Irvine (November 1974).

23 Schwartz, J. T. Automatic Data Structure Choice in a Language of Very High Level. To appear Proc. Second ACM Symposium on Principles of Programming Languages, Palo Alto, CA (January 1975).

24 Shneiderman, B. Towards a Theory of Encoded Data Structures and Data Translation. Technical Report #13, Computer Science Department, Univ. of Indiana (July 1974).

25 Shneiderman, B. and P. Scheuermann. Structured Data Structures. Comm. of the ACM, vol. 17, no. 10 (October 1974), pp. 566-574.

26 Taft, E. A. and T. A. Standish. PPL User's Manual. Aiken Lab, Harvard University (January 1971).

27 Tonge, F. M. List Processing Techniques and Languages -- Design Decisions. Proc. Univ. of Michigan Engineering Summer Conf. (1967).

28 Turski, W. M. A Model for Data Structures and its Applications. ACTA Informatica, vol. 1 (1971), pp 26-34.

29 Weizenbaum, J. Symmetric List Processor. Comm. of the ACM, vol. 6, no. 9 (September 1966), pp. 524-543.

30 Wegbreit, B. The ECL Programming System. Proc. AFIPS FJCC, vol. 39 (1971), pp. 253-262.

31 van Wijngaarden, A. et al. Report on the Algorithmic Language ALGOL68. Numerische Mathematik, vol. 14, no. 1 (February 1969), pp. 79-218.

32 Wirth, N. The Design of a PASCAL Compiler. Software -- Practice and Experience, vol. 1, no. 4 (1971), pp. 309-333.

33 Wirth, N. The Programming Language PASCAL (Revised Report). Berichte der Fachgruppe Computer-Wissenschaften, Eidgenossische Technische Hochschule, Zurich (December 1973).

APPENDIX

The Modelling Structures Formalism (reference 22 of this paper), technical report #52, is available form the Department of Information and Computer Science, University of California, Irvine.