Copyright © 1974, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

AN OVERVIEW OF THE VERS2 PROJECT:
HIGH LEVEL LANGUAGES IN AUTOMATIC PROGRAMMING

by

Jay Earley

Memorandum No. ERL-M416

December 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
Computer Science Division
University of California, Berkeley
94720

AN OVERVIEW OF THE VERS2 PROJECT:

HIGH LEVEL LANGUAGES IN AUTOMATIC PROGRAMMING

Jay Earley

Computer Science Division, of the
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

ABSTRACT

In the VERS2 project, our approach to the area of automatic programming is through the development of high level languages (which allow a solution to a problem to be stated more easily), and optimizers (this is meant very broadly) for these languages which make their programs as efficient as ones written (with more difficulty) in a lower level language. The specific programming language components which we deal with are data structures (and complex operations on them), naming structure, and extensibility. We are developing and experimenting with a particular high level language VERS2, which embodies these concepts.

This research was supported under NSF Grant #GJ 34342X.

INTRODUCTION

Automatic programming is a field of endeavor in which we strive to automate the work that is normally done by programmers, with the ultimate goal of allowing computers to be used by people who are not necessarily trained in programming and allowing computers to be more effectively used by trained programmers. We feel that one important approach in this area is the automation of programming by developing high level languages which allow a solution to a problem to be stated more easily, and optimizers (we mean this very broadly) for these languages which will make their programs as efficient as ones written (with more difficulty) in a lower level language. In fact, we feel that there is a continuum of levels of description between the statement of a problem and the statement of an efficient solution to a problem, and we are working upwards along this continuum.

In [1] Balzer presents a model of an automatic programming system which is quite useful in describing where our work fits into the field. His model has four phases:

1) problem acquisition, in which the system learns from the user a statement of the problem to be solved;

2) process transformation, in which a method for solving the problem is generated;

3) automatic coding, in which an efficient program is constructed from the description of the method; and

4) verification, in which the system's understanding and execution

of the user's intent is checked.

OUTLINE OF RESEARCH

Our work falls into two areas, 1) development of the language (concepts, not syntax), which would be the interface between the various phases in Balzer's model, and 2) work on the automatic coding phase. Let us first explore what we mean by the interface language. There should be some set of semantic concepts which can be used to state both a problem and a method for solving it in terms which are as close to the domain of the problem and as independent of efficiency considerations as is required. We call this set of concepts the "interface language" because it would serve as the form for the output of the problem acquisition phase, the input and output of the process transformation phase, and the input to both the automatic coding and verification phases. Clearly, the design of this language will have a profound influence on the entire system. One main part of our work will be the development of an interface language which allows the appropriate naturalness and ease of description of problem domains, problems, and methods of solution.

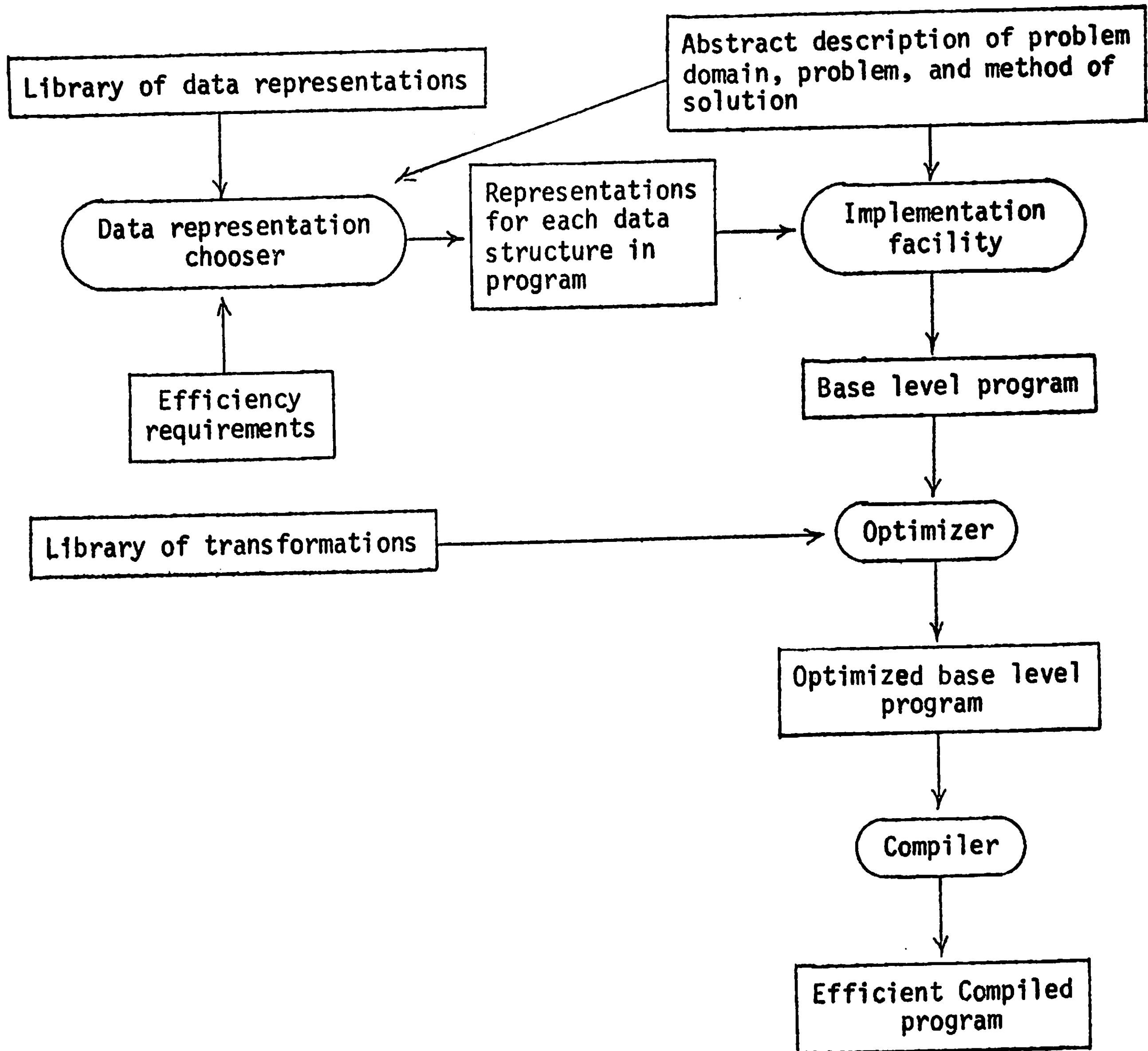
In order to explain more fully what we mean by automatic coding, we formulate three development stages for an automatic programming system.

Stage 1 (conventional): The user supplies the algorithms and data definitions, and makes essentially all implementation decisions.

Stage 2 (partially automatic): The user supplies the algorithm (perhaps partially), and the system completes it and derives an efficient implementation.

Stage 3 (automatic): The user supplies the problem and the system derives the algorithm and its implementation.

We believe that there is a continuum between the stages and that work on later areas will likely proceed concurrently with work on earlier ones. Our work consists of transforming a system from Stage 1 to Stage 2. We can show this development in more detail as follows:



This is a diagram of the principal parts of our Stage 2 system. Our Stage 1 system differs from it only in that some circles are missing and others are performed by the user rather than by the system. The user provides, at

the top, an abstract description of his problem domain (if someone else has not already done this), his problem, and a method for solving it. This is then combined with descriptions of representations for each data structure in the program (also written by the user in our Stage 1 system) by the implementation facility to produce a base level program. This is a program in a higher level language which reflects the choices of data structure representation that have been made. Notice that in the Stage 1 system, the data representation chooser is not used at all. The next step is to apply the optimizer to effect certain rearrangements and deletions of code to improve the performance of the base language program. In the Stage 1 system the optimizer will have a fixed set of transformations that it can apply (it will not use a library) and they will be rather simple ones.

The final step is to compile the program. We would like to emphasize that the program can be executed and debugged or tested at any step in its development, including the original abstract description.

In progressing to a Stage 2 system, we add the following things: 1) a library of data structure representations from which the user may choose the one appropriate for his program; 2) an automatic mechanism for choosing representations from this library; 3) a modification to the optimizer so that it may be table-driven; that is, it would use a library of transformations which could be augmented by the user.

DETAILS OF OUR APPROACH

We will now describe our ideas in more detail. We have been designing and working with an abstract level of data structure description and manipulation which we call the "relational" level.[2] It is intended to be used in a programming language in order to allow the programmer to express his problem solutions in terms which are natural to his problem domain -- or

at very least, in terms of the abstract data structures which are most relevant to his needs rather than the machine-oriented structures currently used. The specific structures we use are sets, relations, sequences, and tuples. In addition to these, we provide a collection of powerful operations called iterators, which contribute to the elimination of implementation details in the expression of algorithms. These are abbreviations for what would normally be written out as various kinds of loops. We have a wide variety of them including quantifiers, operations which form the set of all objects satisfying a certain condition, and minimization and sorting operations (see [4]). Other languages containing some of our high level data structure and operations are SETL[12], SAIL[13], MADCAP[14], and QA4[15].

We would like to take programs written in a programming language using these data structures and operations, and automatically generate efficient representations for the data structures and optimized versions of the programs. This process will be done in a number of stages. First, we provide our language with an implementation facility. This includes having data structures in the language which are relevant to representation of structures in machines (vectors and structures as in ECL[7]). It also includes a facility whereby one may specify the representation of the abstract data structures using a "structured programming" approach[3]. For each use of a data structure type, the programmer may indicate that it is to be represented as some lower-level type, and then he may rewrite the relevant primitive operations which are used on the higher-level type as subroutines in terms of the lower-level type. This may be repeated until things are expressed in terms of machine-level structures.

Next, we will get some experience at using this kind of a system, and develop a library of representations for each type of structure and a

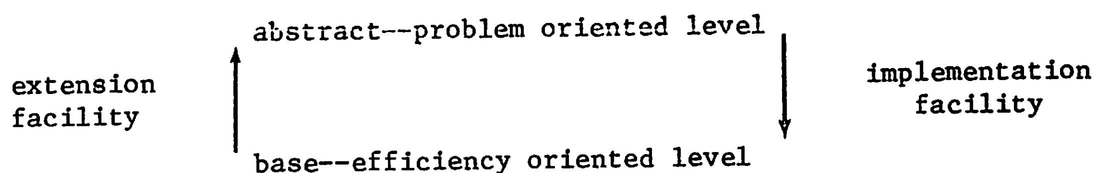
characterization of the circumstances under which each representation is most useful. This would include information about the cost of the representation in space (per element of the structure) and the cost in time of applying each primitive operation to the structure using this representation. We will then try to automate the choice of data structure representation given information about the average size of the structure, relative frequency of applying the various primitive operations to the structure (e.g., frequency of adding to a set as opposed to testing for set membership), and the user's requirements for space and time efficiency.

As an example, consider the problem of programming a computer to play cards. At the abstract level, a "card" is a pair (2-tuple) consisting of a "suit" and a "denomination". A "pile" is a sequence of cards, and a "hand" is a set of cards. An iterator which might be used in such a program could calculate the set of all cards C in a hand H which were of a particular suit S:

$$\{C \in H \mid SUIT(C) = S\} \quad .$$

In the system-provided default implementation for the iterator we would generate each member of S and test its suit. However, there will be a library implementation for a set which represents it as a relation mapping some domain (suits in this case) into disjoint subsets of the original set. This implementation might be chosen in this case because it makes the above iterator quite efficient. Implementations for the relation and for the subsets would be chosen as well. The relation could be a 4-tuple and the subsets could be linked lists. Of course, in reality, a hand of cards is so small that it might not even matter what these representation choices were, but they illustrate how the system works.

Notice that this approach is significantly different from an ordinary extensible language approach. We need to have an extensible language, of course, because the abstract data structure language is, in fact, an extension of a simpler base language. However, there are two important differences: 1) The user works down from the top, not up from the bottom. He writes an abstract program first, debugs it using default representations provided by the system, and then worries about the lower levels of his program. He must be able to easily modify the default extensions to use his own (or the library's), which are more efficient. This requires additional mechanisms which we are calling the implementation facility. 2) A great deal of research has gone into determining which constructs should be in the high level abstract language. This is every bit as important as the rest of the system, for without the right input from the user, the best implementer in the world will be relatively ineffective. Our approach can be summarized in this diagram:



We believe that the method of choosing an implementation from a library will work reasonably well with programs which do not rely heavily on iterators. For those programs which are written at a high enough level to use iterators as a primary means of specification, methods for automatically designing data structure representation (based on the particular iterators used) will be necessary. We have one significant method already (called interator inversion) for designing the data structure representations required to convert a batch algorithm to an incremental one using the

information contained in the iterators. We have also developed a comprehensive enought set of iterators[17] that some complicated algorithms can be expressed entirely in terms of applicative programs using iterators. This is a form of highly structured programming, and we feel that it opens the way to more powerful methods of automatically improving programs. Iteration inversion is the first example of such a method but we expect to develop more.

Orthogonal to this data representation research, there will be an effort in code optimization. Where the previous section dealt with the form of data structures, this will deal with the form of the program. Its object will be to eliminate and rearrange code to produce a more efficient program. It objectives, then, will be the same as conventional efforts in code optimization and will rest heavily on existing technology. The addition problem which we face is attempting to do this in the presence of complex data structures and a rich collection of operators.

One can view the approach to global code optimization in two parts: gathering information and applying transformations. This is not to say that the two happen serially; in fact, there will probably need to be a great deal of interaction between the two phases. The presence of complex data structures affects both phases. The information gathering is different in that modifications to complex data structures (especially those involving pointers) affect our knowledge of the program differently than assignments to simple variables and arrays. The transformations to be applied are also different in that much of the code deals with data structures rather than arithmetic expressions. In particular, we believe that embedding knowledge about the algebraic structure of operations on data structures in the

information contained in the literature. We have also developed a complete [illegible] specification [illegible]. The more complicated algorithms can [illegible] programs using iterations. [illegible] we feel that it opens the way [illegible] methods of automatically improving programs. [illegible] method [illegible] to [illegible]

Orthogonal to this data representation research, there will be an effort [illegible]. Where the previous research deals with the types of data structures, this will deal with the form of the program. The object will be to alter and rearrange code to produce a more efficient program. [illegible] conventional efforts in code optimization [illegible] rely heavily on [illegible] technology. The problem which [illegible] is attempting to do this in the presence of complex data structures and [illegible] operators.

[illegible] approach to [illegible] code optimization [illegible] information and applying [illegible]. This is not [illegible] can happen [illegible]. In fact, there will probably need to be a great deal of interaction between the two phases. The presence of complex data structures affects both phases. The information gathered is different [illegible] complex data structures [illegible] of the program [illegible]. The [illegible] In particular, we believe [illegible]

[illegible]

optimizer will enable it to remove many redundancies. In addition, much of our code will have been generated at least semi-automatically by the data representation phase and therefore will probably contain more redundancies.

Our optimizer will be constructed in a table driven fashion so that it can accept new transformations plus hints about when they would be useful. This will be especially necessary because of the lack of experience in using optimizers with data structures.

In addition to this basic thrust of our research, we have been exploring two other areas which are important to the development of high level programming languages: Language extension and naming structure. Our effort in language extension has three parts: (1) We have developed a method of syntax description[5] which separates ordinary issues of syntax from precedence issues, resulting in grammars that are smaller, easier to construct and read, and easier to extend. (2) We have developed an LR(1) parse constructor which handles all LR(1) grammars yet produces parse tables at about the same size as Simple LR(1) constructors[16]. (3) We have developed a method of specifying the meaning of extensions[6] which involves less implementation detail and fewer special constructs in the language.

Naming structure establishes a correspondence between names in a program and values. We have developed a new form of naming structure, which we call "module structure"[8] which is more general and flexible than existing schemes, such as Algol's "block structure" or LISP's "dynamic naming". It allows separately written routines or program segments to be instantiated and combined in a variety of ways to enhance a programmer's ability to modularize his task.

CURRENT PROGRESS

We are designing and implementing a programming language called VERS2[10]

[illegible] enable us to remove many redundancies. In addition, much of [illegible] will have been generated at least semi-automatically by the system [illegible] phase and procedures will thereby contain more information.

Our retrieval program will be [illegible] driven fashion so that it can accept and [illegible] about when they would be useful. This will be [illegible] necessary because of the lack of correspondence in [illegible] with data structures.

In addition to this main thrust of our research, we have been exploring two other areas which are important to the development of high level programming languages: language extension and naming structures. Our efforts in language extension [illegible] these areas: (1) We have developed a method of syntax description[8] which separates ordinary issues of syntax from precedence issues, resulting in grammars that are smaller, easier to construct and read, and easier to extend. (2) We have developed an LR(1) parser constructor which handles all LR(1) grammars yet produces parse-tables as about the same size as simple LR(1) constructors[10]. (3) We have developed a method of specifying the meaning of extensions[9] which involves less [illegible] detail and fewer special constructs in the language.

[illegible]

[illegible] We have developed a new form of naming structure, which we call "module structure"[11] which is more general and flexible than existing schemes, such as Algol's "block structure" or Simula's "dynamic scoping". It allows [illegible] routines or program segments to be instantiated and [illegible] of ways to enhance a programmer's ability to [illegible] his task.

RESEARCH PLANS

[illegible] designing and implementing a programming language called [illegible]

in order to test out the ideas expressed so far. We are implementing VERS2 as an extension of ECL on a PDP10 at USC's Information Sciences Institute over the ARPA network. At the current time, a version of VERS2 containing the high level data structures and operations, and the implementation facility is just becoming operational. Our next step is to start developing a library of representations and to start thinking seriously about choosing data structure representations and about optimization. An extension facility has been designed integrating the three components mentioned earlier which we expect to be running in a couple of months. We expect to make this extension facility generally available, not just for extending VERS2, but also for extending ECL or any language implemented in ECL. The naming structure is completely designed but cannot be implemented until certain improvements are made to ECL which we expect sometime in the next year. A Harvard group under Cheatham is in the process of developing tools for use in optimization of ECL programs [9], and we expect to collaborate with them on this effort.

In addition to the publications mentioned so far, we have produced a short note on language design [11], and we anticipate producing a paper on extensible data types soon.

in order to flesh out the ideas expressed so far. We are implementing VERS2 as an extension of ECL on a PDP10 at USC's Information Sciences Institute over the ARPA network. At the current time, a version of VERS2 containing the high level data structures and operations, and the implementation facility is just becoming operational. Our next effort is to start developing a library of representations, to experiment with [illegible] choosing data structure representations and [illegible] optimization. An extension facility has been designed integrating the [illegible] mentioned earlier which we expect to be running in a couple of months. We expect to make this extension facility generally available not just for extending VERS2, but also for extending ECL or any language implemented in ECL. The [illegible] structure is completely designed but cannot be implemented until certain improvements are made to ECL which we expect sometime in the next year. A Harvard group under Cheatham is in the process of developing tools for use in optimization of [illegible] programs[4], and we expect to collaborate with them on this effort.

In addition to the publications mentioned so far, we have produced a short note on language design[11], and we anticipate producing a paper on extensible data types soon.

ACKNOWLEDGEMENTS

I have been assisted in the design and implementation of VERS2 by Steve Chernicoff, Dan Carnese, Jeff Barth, and Murray Bowles. The LR(1) parse constructor has been developed by Steve Meyer. Larry Barnes has provided many valuable ideas on the direction of the project and the organization of this paper.

REFERENCES

[1] Balzer, R. M., "Automatic Programming," USC Information Sciences Institute, 1972.

[2] Earley, J., "Relational Level Data Structures for Programming Languages," to appear in Acta Informatica.

[3] Dijkstra, E. W., "Structured Programming," Software Engineering Techniques, NATO Science Committee, 1970.

[4] Earley, J., "High Level Operations in Automatic Programming," to be presented at SIGPLAN Simposium on Very High Level Languages, March 1974.

[5] Earley, J., "Ambiguity and Precedence in Syntax Description," Technical Report.

[6] Earley, J., "Syntax Extension Using a Run Time Model," Technical Report.

[7] Wegbreit, B. et al., ECL Programmer's Manual, Harvard University, 1972.

[8] Earley, J., "Naming Structure and Modularity in Programming Languages," Technical Report.

[9] Chetham, T. and Wegbreit, B., "A Laboratory for the Study of Automating Programming," Proc. SJCC, 1972.

[10] Carnese, D., et al, VERS2 Manual, in preparation.

[11] Earley, J., "Initialization and Null Objects in Programming Languages," Technical Report.

[12] Schwartz, J.T., "Abstract Algorithms and a Set-Theoretic Language for Their Expression," New York University, 1970-71.

[13] Feldman, J. H., et al, "Recent Developments in SAIL," Proc. FJCC, 1972.

[14] Wells, M. B. and Morris, J. B., "The Unified Data Structure Capability in Madcap VI," International Journal of Computer and Information Sciences, September, 1972.

[15] Derksen, J., Rulifson, J.F. and Waldinger, R.J., "The QA4 Language Applied to Robot Planning," Stanford Research Institute, 1972.

[16] DeRemer, F.L., "Simple LR(k) Grammars," Comm. ACM, 14, July 1971.

[17] Earley, J., "High Level Iterators and a Method for Automatically Designing Data Structure Representation," in preparation.