SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 5, 269–278 (1975)

A Transportable Extendable Compiler*

VICTOR R. BASILI AND ALBERT J. TURNER
Computer Science Department, University of Maryland, College Park, Maryland, U.S.A.

SUMMARY

This report describes the development of a transportable extendable self-compiler for the language SIMPL-T. SIMPL-T is designed as the base language for a family of languages. The structure of the SIMPL-T compiler and its transportable bootstrap are described. In addition, the procedures for generating a compiler for a new machine and for bootstrapping the new compiler on to the new machine are demonstrated.

KEY WORDS Transportable Extendable Compiler Bootstrapping SIMPL-T SIMPL family

INTRODUCTION

The differences in computer architecture and in operating systems make the development of a transportable compiler for a programming language a formidable task. This paper describes the development of a reasonably transportable and extendable compiler for the language SIMPL-T.[1]

Most compilers that are designed to be transportable are self-compiling; that is, they are written in the language that they compile. The NELIAC compilers[2] were among the first self-compiling compilers, and more recent efforts include the XPL[3,4] and BCPL[5] compilers. The effort required to transport these compilers includes the rewriting of the code generation portion of the compiler to generate object code for the new machine and the design and programming of run-time support routines. An existing implementation can then be used for the debugging and generation of a compiler for the new machine. As an alternate procedure, the BCPL design allows the bootstrap process to be performed without using an existing implementation by writing (and debugging) two code generators, one in BCPL and another in an existing language already implemented on the target machine.

The SIMPL-T compiler is also self-compiling and the effort required to transport it to a new machine consists of the design and programming of a new code generator and a run-time environment for SIMPL-T programs executing on the new machine. This paper discusses three features of the transportable, extendable SIMPL-T compiler.

Firstly, there is a transportable bootstrap which permits the SIMPL-T compiler to be transported to most machines without using an existing implementation of the language. Moreover, this bootstrap requires no extra effort such as writing a temporary code generator for the bootstrap that will not be used in the final implementation on the new machine. This transportable bootstrap distinguishes the SIMPL-T bootstrap procedure from that required for most other self-compiling compilers.

* This research was supported in part by the Office of Naval Research under Grant N00014-67-A-0239-0021 (NR-044-431) to the Computer Science Center of the University of Maryland, and in part by the Computer Science Center of the University of Maryland.

Received 16 January 1974
Revised 21 October 1974

© 1975 by John Wiley & Sons, Ltd.

Secondly, the highly modular design of the compiler, along with the features of the SIMPL-T language itself, minimizes the effort required to write and interface the new code generator and run-time environment. A reasonably competent systems programmer should be able to bootstrap SIMPL-T to a new machine in one to three months. The actual time required depends mostly on the quality of the object code to be produced by the compiler.

Finally, the compiler has been designed to permit extensions so that other compilers may be built out of it.

THE SIMPL-T LANGUAGE

SIMPL-T is a member of the SIMPL family of structured programming languages.[6] The SIMPL family is a set of languages each of which contains common features, such as a common set of data types and control structures. The fundamental idea behind the family is to start with a base language and a base compiler and then to build each new language in the family as an extension to the base compiler. Thus, each new language and its compiler are bootstrapped from some other language and compiler in the family.

SIMPL-T was designed to be the transportable extendable base language for the family. The transportable extendable base compiler for SIMPL-T was written in SIMPL-T to permit the entire family of languages to be implemented on various machines in a relatively straightforward manner, as suggested by Waite.[7] (The extensibility scheme is thus similar to that used for Babel and SOAP.[8])

Other members of the SIMPL family include a typeless compiler-writing language, SIMPL-X,[9] a standard mathematically-oriented language, SIMPL-R[10], a systems implementation language for the PDP-11, SIMPL-XI[11] and the graph algorithmic language GRAAL.[12] The original design and implementation of the SIMPL family of languages and compilers were done at the University of Maryland for the UNIVAC 1100 series computers.

SIMPL-T and other members of the SIMPL family have been used in research projects and in classes at a variety of levels in the Computer Science Department at the University of Maryland. SIMPL-T is being used as an implementation language by the Defense Systems Division, Software Engineering Transference Group at Sperry Univac. SIMPL-R is being used in the development of a transportable system for solving large spare matrix problems.[10]

The salient features of SIMPL-T are

(1) Every program consists of a sequence of procedures that can access a set of global variables, parameters or local variables.

(2) The statements in the language are the assignment, if-then-else, while, case, call, exit and return statements. There are compound statements in the language, but there is no block structure.

(3) There is easy communication between separately compiled programs by means of external references and entry points.

(4) There is an integer data type and an extensive set of integer operations including arithmetic, relational, logical, shift, bit and partword operations.

(5) There are string and character data types. Strings are of variable length with a declared maximum. The range of characters is the full set of ASCII characters.

The set of string operators includes concatenation, the substring operator, an operator to find an occurrence of a substring of a string and the relational operators.

(6) Strong typing is imposed and there are intrinsic functions that convert between data types.

(7) There is a one-dimensional array data structure.

(8) Procedures and functions may be recursive but may not have local procedures or functions. Only scalars and structures may be passed as parameters. Scalars are passed by value or reference and structures are passed by reference.

(9) There is a facility for interfacing with other languages.

(10) There is a simple set of read and write stream I/O commands.

(11) The syntax and semantics of the language are relatively simple, consistent and uncluttered.

It seems prudent to emphasize here that SIMPL-T programs are not necessarily transportable. The language contains some highly machine-dependent operations, such as bit manipulation operators. The merits and disadvantages of having such operations in the language will not be discussed here. However, it is not difficult to write SIMPL-T programs that are transportable, and this is what was done in writing the SIMPL-T compiler.

A simple stack is adequate for the run-time environment in an implementation of SIMPL-T. This together with the simple I/O facilities in the language and the lack of reals makes the design and implementation of support routines easier than for languages such as FORTRAN and ALGOL.

The availability of external procedures in SIMPL-T means that operating systems interfaces that may be desired for a compiler can easily be managed by writing the interface as an external procedure. Such external interfaces are needed only for uses involving individual operating system idiosyncracies, however, as SIMPL-T is sufficiently powerful to allow the compiler to be written entirely within itself. (Examples of such uses are the obtaining of date and time, the interchanging of files, etc.)

THE SIMPL-T COMPILER

Although SIMPL-T programs can be compiled in one pass, the compiler was written as a three-pass compiler with separate scan, parse and code generation phases. The separate code generator is needed for the portability scheme, and separate scan and parse phases promote modularity and provide more flexibility for implementing later extensions.

The scanner and parser are designed and programmed to be machine independent so that the compiler can be transported to a new machine by writing only the code generation pass for that machine. The parser generates a file containing a machine-independent intermediate form of a SIMPL-T program that can readily be converted into machine code for most computers. (This approach is similar to that used for the BCPL compiler.)

Extendability in the scanner and parser is provided by using a modular approach that avoids the use of obscure programming 'tricks'. In order to enhance the clarity and ease of extendability, occasional inefficiency and repetition of code has been allowed. The parser uses a syntax-directed approach that is based on an optimized SLR(1)[13] algorithm and uses an operator precedence[14] scheme for parsing expressions.

An additional optimization pass is planned that will perform machine-independent optimization on the intermediate output from the parser. (Such an optimizer was written

for an earlier version of the compiler but has not been updated for the latest version.) The design of the compiler permits the use of a variety of machine-independent optimization techniques, such as those suggested by Hecht and Ullman,[15] and Kildall.[16] In order to provide more efficient usage of storage on a variety of machines, the scan and parse phases of the compiler are written in macro code. A macro preprocessor[17] is used to generate different versions of these phases for different word sizes on the target machines. The differences mostly involve the symbol table, whose entries consist of several 16-bit fields. For machines having a word size of less than 32 bits, these fields are allocated one per word; for larger words, one field is right-justified in each halfword.

All implementation-dependent decisions in the compiler are delayed until the code generation phase. These include the assignment of addresses, decisions on immediate constants, generation of object output for initialized variables and the handling of entry points and external references. These actions could be performed more efficiently during the scan phase, but delaying them until code generation facilitates a new implementation of the compiler.

The intermediate form generated by the parser is a quadruple[18]

OP, A, B, R

consisting of an operation field, an A-operand, a B-operand and a result field. The quads represent high-level operations that make no assumptions about the architecture of the machine for which the compiler is to generate code. Some redundancy is introduced into the quads so that writing a straightforward code generator is made easier.

The quads are generally of two types: operation quads and structure quads. The operation quads correspond to the primitive operators of the SIMPL-T language, and the structure quads represent the program structure. As examples, the operation X + Y would be represented by the quad

+, X, Y, t

where t is an internal designator for the result; a statement beginning

IF X > Y THEN

would generate the quads

>, X, Y, t

IF, t, ,

The choice of quads over a polish string representation[18] was made primarily to enhance the writing of a machine-independent optimization pass. Quads also allow more flexibility in the design of a code generator since, for example, no stack is required. Quads were chosen over two-address codes (triples)[18] for the same reasons, although the same arguments apply to a lesser degree. We believed that there would be less bookkeeping effort required for quads than for triples. Our experience thus far has shown the choice of quads to be satisfactory in every way.

The high level of the quads allows a great deal of flexibility as to the efficiency of the object code generated. For example, the original 1108 code generator, designed and implemented in three weeks, was fairly straightforward and generated mediocre to poor object code. However, an extensive revision of the code generator, requiring a six-week effort, yielded a compiler that provides good object code comparing favourably with the code that is produced by other compilers on the 1108. Thus, the time and effort expended on a

new implementation of SIMPL-T depends a great deal on the quality of the object code to be produced for the new machine.

Table I gives a comparison of the core requirements for the ALGOL, FORTRAN and SIMPL-T compilers on the UNIVAC 1108. The FORTRAN figures are for the smaller of the two standard FORTRAN compilers supported by UNIVAC, and the ALGOL compiler used is the NUALGOL compiler from Norwegian University. Both the ALGOL and FORTRAN compilers are coded in assembly language.

Table I. Size comparisons for UNIVAC 1108 compilers. K = 1,000 words

Compiler	Overlay segments	Space required			Non-overlayed size		
		Instructions	Data	Total	Instructions	Data	Total
ALGOL	3	13K	9K	22K	30K	17K	47K
FORTRAN	6	16K	19K	35K	53K	39K	92K
SIMPL-T	4	15K	14K	29K	30K	20K	50K

Table II. Comparison between a sample program coded in FORTRAN and SIMPL-R. The timings are CPU times, and the program sizes include library routines

Language	Compile time	Object program size		Execution time
		Instructions	Data	
FORTRAN	6·9 sec	6,873	21,862	7·7 sec
SIMPL-R	7·0 sec	5,339	20,875	6·6 sec

Comprehensive comparisons have not been made between object programs produced by the different compilers. However, the results of one comparison between the object programs generated by the FORTRAN and SIMPL-R compilers is given in Table II. (The SIMPL-R compiler is an extension of the SIMPL-T compiler and the two compilers generate identical code for SIMPL-T programs.) For this comparison, a sparse matrix problem was coded in both FORTRAN and SIMPL-R and executed on several sets of data.[10] Both programs consisted of about 750 source cards (360 SIMPL-R statements), and the execution timings are for a typical set of test data.

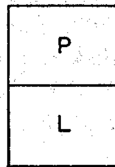
The performance figures in Tables I and II illustrate some success in achieving the SIMPL-T design criterion of generating efficient object code. The favourable comparisons are in spite of the fact that the FORTRAN compiler has a good optimizer, while the SIMPL-T and SIMPL-R compilers have only local optimization.

The figures also show reasonable results in compile time for the SIMPL compilers when compared with FORTRAN. This is in spite of the facts that the SIMPL compilers are designed for portability rather than for fast compilation and are coded in a high-level language rather than in assembly language.

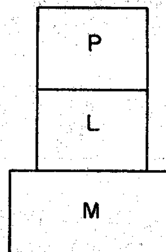
BOOTSTRAPPING SIMPL-T

Plans for transporting a compiler from computer M to a new computer N must include a procedure for bootstrapping on to the target machine N unless the compiler is written in a language that already exists on the target machine. Since the SIMPL-T compiler is written in SIMPL-T, a bootstrap is required in order to transport the compiler.

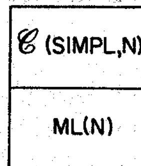
Two procedures for bootstrapping SIMPL-T on to a new machine are illustrated in Figures 1 and 2. The notation



denotes program P coded in language L and



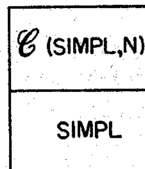
denotes program P, in language L, executing on machine M (so that L would be machine language for M). $\mathscr{C}(L, M)$ denotes a language L compiler for machine M, and ML(M) denotes machine language for machine M. Thus the objective of a bootstrap of SIMPL-T to a new machine N is to obtain



Finally, $\mathscr{T}(L_1, L_2)$ denotes a translator from language L_1 to language L_2, and

indicates that A is input to processor B and the output is C.

It is worth noting that the code generation module of



represents the major effort required to transport the SIMPL-T compiler to a new machine N.

One method of bootstrapping that could be used for SIMPL-T is to compile the new compiler for machine N using the existing SIMPL-T compiler on machine M and then transport the object code to the new machine. This procedure, illustrated in Figure 1, has

the advantage that no intermediate language is involved, and it is possibly the best procedure to use if a system that supports an existing SIMPL-T compiler is conveniently available.

As an alternative to using an existing SIMPL-T compiler for the bootstrap, and as a means of bootstrapping SIMPL-T on to our 1108 initially, it was decided to write a transportable bootstrap compiler. This required that the bootstrap compiler be written in a transportable language and that the compiler produce transportable output.

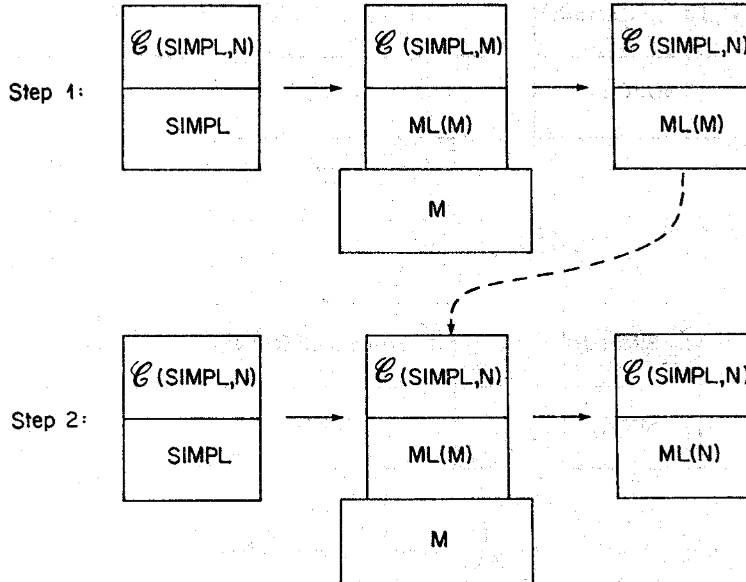


Figure 1. Bootstrapping a SIMPL compiler on to a machine N using an existing implementation on machine M

Of the languages available only FORTRAN and SNOBOL satisfied the main requirements of portability and availability. SNOBOL was preferred because of its recursion and string handling facilities, but the lack of compiler versions of SNOBOL is a disadvantage for several reasons.[19] SNOBOL interpreters are usually large and slow and are not designed for easily debugging large modular programs.

On the other hand, FORTRAN provides convenient facilities for working with separately compiled modules, but it is undesirable for writing portable string manipulation programs. It was thus desired to find a solution that would provide the ease of programming a translator in SNOBOL and the ease of working with programs written in FORTRAN.

The solution obtained was to write a translator in SNOBOL4 that translates a SIMPL-T program into ANSI FORTRAN IV. This would yield a bootstrap procedure that would enable SIMPL-T programs to be run on a machine that has no SIMPL-T compiler, provided the machine has SNOBOL4 and FORTRAN IV available. The SNOBOL bootstrap translator would be used to convert a SIMPL-T program into a FORTRAN program, and the FORTRAN program could then be compiled and executed. This procedure is illustrated in Figure 2.

To facilitate the use of the bootstrap, string handling and I/O packages (written in FORTRAN) are included. Thus the only effort required to transport the bootstrap (in addition to the effort required for the compiler) is to write a few machine-dependent subroutines, such as bit manipulation and system interface subroutines. This practically negligible effort yields the desired bootstrap package for a new machine.

It should be noted that the SNOBOL translator produces transportable FORTRAN code through such devices as allocating strings one character per word. Essentially all of the features of SIMPL-T are supported by the translator, including recursion, call by value and reference, and externals.

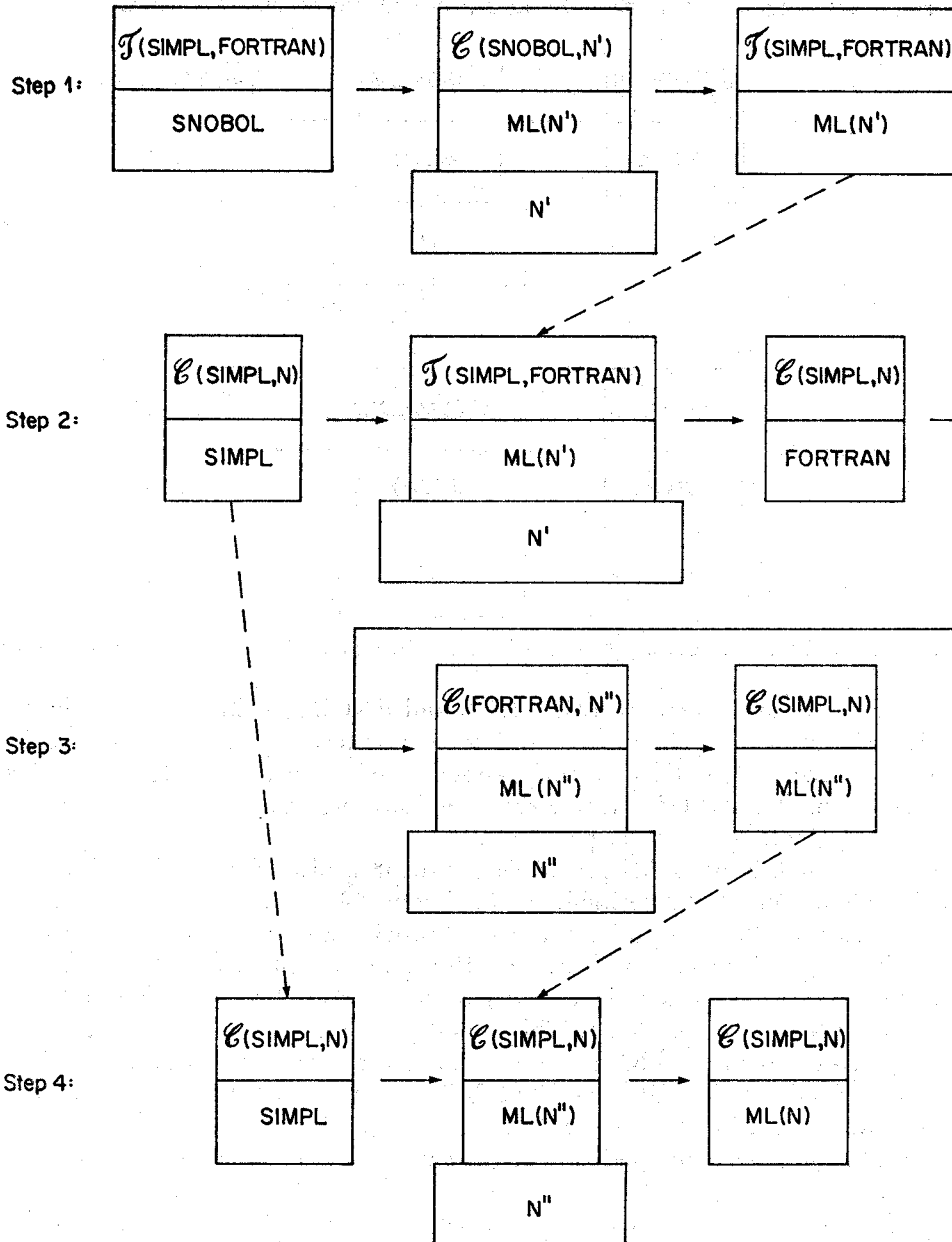


Figure 2. Bootstrapping a SIMPL compiler on to machine N using the SNOBOL translator. Machines N′ and N″ would normally (but need not) be the same as machine N. Note that if N″ = N, the last step is still needed to produce a more efficient compiler. Note also that Steps 1 and 2 would be combined if a SNOBOL interpreter (instead of a compiler) were used

Some variation on the bootstrap procedure using the SNOBOL translator may be desirable if SNOBOL, FORTRAN or both are not available on the target machine. Either the translation of a SIMPL-T program into FORTRAN, or the compilation and execution of the resulting FORTRAN program (or both) could be done on another machine. (This might be the case, for example, in bootstrapping to a small machine for which SNOBOL is not available.) Thus, the bootstrap process is rather flexible due to the portability of the SNOBOL translator and of the FORTRAN programs that it produces.

RESULTS AND COMMENTS

The bootstrap procedure described here was used initially to bootstrap the typeless language SIMPL-X on to the UNIVAC 1108 at the University of Maryland Computer Science Center. This bootstrap was facilitated by the fact that the variables of SIMPL-X translated directly into FORTRAN integer variables.

A code generator for the PDP-11 has also been written in order to implement the systems programming language SIMPL-XI mentioned earlier. This code generator was interfaced with the existing scanner and parser with no problems. SIMPL-XI, which also required some extensions to the compiler, is being run as a cross-compiler on the 1108 for the PDP-11.

The SIMPL-T compiler was bootstrapped from SIMPL-X and has been extended to yield a compiler for SIMPL-R, a language that has reals. The SIMPL-R implementation[10] was a six-week effort by a programmer who was not familiar with either the SIMPL-T compiler or the 1108 computer and operating system.

Currently, efforts are under way to bootstrap SIMPL-T on to the IBM 360/370 machines. The SNOBOL–FORTRAN bootstrap for SIMPL-T was recently completed and has been used to run the scan and parse passes of the compiler on a 360.

While the bootstrap procedure has been successful in general, there have been some problems. No compiler version of SNOBOL was available for the 1108, and the available interpreter versions proved to be inadequate and required local modification. SPITBOL on the 360 has been a vast improvement and would have more than adequately solved this problem had a working version been available for the 1108.

The other problems were primarily due to the inadequacies and restrictions of FORTRAN. Again, if SPITBOL were generally available, most of these problems could have been eliminated by translating SIMPL-T into SNOBOL (SPITBOL). This would have made available such features as recursion and string data, thereby facilitating the translation.

Although these problems were foreseen, they were underestimated. The large amount of time and memory required for the SNOBOL programs and the size of the FORTRAN programs generated (about 90K words for the scanner and parser on the 1108) made the development of the bootstrap an expensive and time-consuming process. Furthermore, these requirements make the bootstrap procedure impractical (if not impossible) for small machines.

Yet these were the only languages available for which there was reasonable expectation of producing portable programs. This is a rather sad commentary on the availability of reasonable general-purpose languages and compilers, and indicates a need for widespread implementation of languages and compilers such as SIMPL-T and its compiler.

On the basis of our experience, we believe that this approach to bootstrapping a language on to a variety of machines would be quite satisfactory if a suitable language were already

available on the target machines. Even with the drawbacks mentioned, we know of no alternative that would provide an easier means of performing a stand-alone bootstrap.

ACKNOWLEDGEMENTS

The bootstrap for SIMPL-X was written by Mike Kamrad, and the bootstrap for SIMPL-T was written by Bruce Carmichael. The system routines for the UNIVAC 1108 compilers were written by Hans Breitenlohner. C. Wrandle Barth at Goddard Space Flight Center and Robert Knight at Princeton University are bootstrapping SIMPL-T to a 360.

REFERENCES

1. V. R. Basili and A. J. Turner, *SIMPL-T: A Structured Programming Language*, CN-14, University of Maryland Computer Science Center, 1974.
2. M. Halstead, *Machine-independent Computer Programming*, Spartan Books, Rochelle Park, New Jersey, 1962.
3. W. M. McKeeman, J. J. Horning and D. B. Wortman, *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.
4. G. Leach and H. Golde, 'Bootstrapping XPL to an XDS sigma 5 computer', *Software—Practice and Experience*, **3**, No. 3, 235–244 (1973).
5. M. Richards, 'BCPL: a tool for compiler writing and system programming', *AFIPS Proceedings*, **34**, 557–566 (SJCC 1969).
6. V. R. Basili, *The SIMPL Family of Programming Languages and Compilers*, TR-305, University of Maryland Computer Science Center, 1974.
7. W. M. Waite, 'Guest editorial', *Software—Practice and Experience*, **3**, No. 3, 195–196 (1973).
8. R. S. Scowen, 'Babel and SOAP, applications of extensible compilers', *Software—Practice and Experience*, **3**, No. 1, 15–27 (1973).
9. V. R. Basili, *SIMPL-X, A Language for Writing Structured Programs*, TR-223, University of Maryland Computer Science Center, 1973.
10. J. McHugh and V. R. Basili, *SIMPL-R and Its Application to Large Sparse Matrix Problems*, TR-310, University of Maryland Computer Science Center, 1974.
11. R. G. Hamlet and M. V. Zelkowitz, 'SIMPL systems programming on a minicomputer', *Micros and Minis Applications and Design*, Proc. of 9th Annual IEEE COMPCON, 203–206, 1974.
12. W. C. Rheinboldt, V. R. Basili and C. K. Mesztenyi, 'On a programming language for graph algorithms', *BIT*, **12**, No. 2, 220–241 (1972).
13. F. L. De Remer, 'Simple LR(k) grammars', *Comm. ACM*, **14**, No. 7, 453–460 (1971).
14. R. W. Floyd, 'Syntactic analysis and operator precedence', *Jnl ACM*, **10**, No. 3, 316–333 (1963).
15. M. S. Hecht and J. D. Ullman, 'Analysis of a simple algorithm for global flow problems', *ACM Symposium on Principles of Programming Languages*, 1973.
16. G. A. Kildall, 'A unified approach to global problem optimization', *ACM Symposium on Principles of Programming Languages*, 1973.
17. J. A. Verson and R. E. Noonan, *A High-level Macro Processor*, TR-297, University of Maryland Computer Science Center, 1974.
18. D. Gries, *Compiler Construction for Digital Computers*, Wiley, New York, 1971.
19. R. Dunn, 'SNOBOL4 as a language for bootstrapping a compiler', *SIGPLAN Notices*, **8**, No. 5, 28–32 (1973).