F. E. Allen

The History of Language Processor Technology in IBM

The history of language processor technology in IBM is described in this paper. Most of the paper is devoted to compiler technology; interpreters, assemblers, and macro systems are discussed briefly. The emphasis is on scientific contributions and technological advances from a historical perspective. The synergistic relationship between theory and practice is a subtheme.

Introduction

In 1953 IBM introduced an early "automatic-programming" system: Speedcode for the IBM 701 computer. The goal of the system was to [1] ". . . lessen the enormous burdens of the programmer by providing a larger and more convenient instruction repertoire than a given machine provides." In the same paper John Backus and Harlan Herrick go on to state: "There are two principal methods by which automatic-programming systems make these nonmachine operations available to the programmer: the interpretive method and the compiling method." In the almost 30 years that have intervened since these observations were first made, methods for solving "the programming problem" have become more diverse and sophisticated, but the basic problem remains. Language processors—compilers, interpreters, macro systems, and assemblers—are still the principal methods used. In this paper we trace the history of IBM's contributions to the techniques used in today's language processors. The paper concentrates on IBM's scientific and technological contributions, covering only widely used, general purpose techniques or those of particular scientific or historical interest.

The development of language processor technology, particularly compiler technology, is intimately intertwined with the development of computer science. *Ad hoc* solutions to pragmatic problems have continued to be replaced by elegant algorithms. Large, amorphous, seemingly intractable problems have been divided and conquered. The best example of this is in the area of parsing. Early solutions to the problem of decomposing a statement into its component parts were both elaborate and language-specific. The problem became the subject of numerous theoretical investigations and spawned a subfield of computer science: formal language theory. Today, elegant, language-independent parsing systems exist and are a common component of production compilers. The synergistic relationship between theory and practice continues to play a vital role in the development of language processor technology. This relationship is a subtheme of the paper: Particular attention is paid to tracing the origins and evolution of various techniques.

The history of language processors in IBM divides quite naturally into five periods, generally delineated by the introduction of significant new products. The first period, covered in the section on "Early history," starts with the introduction of the IBM 701 in 1952. In 1954 work began on FORTRAN I—a language and system which established the foundations of compiler technology, set standards rarely achieved today, and, as a result, dramatically accelerated compiler development. The second section is devoted entirely to the FORTRAN I compiler. The period from the introduction of the FORTRAN I compiler to the introduction of the System/360—"The late fifties and early sixties"—is the subject of the third section. "The mid- and late sixties" is followed by "Recent history: the seventies," concluding the main presentation. The summary recapitulates the central ideas which have led to the elegant solutions we have today. The paper concludes with some personal observations and predictions.

Early history

• *Historical context*

In 1952 IBM introduced its first production-line electronic digital computer and with it IBM's first language processor—the NR9003 symbolic programming system. Some surprisingly advanced techniques already existed.

Libraries of subroutines and the use of combining routines ("compilers") had been proposed by Goldstine and von Neumann [2] and existed in several systems. A book on programming [3] published in 1951 describes a system having open and closed subroutines and preset and program parameters. Preset parameters were incorporated into the subroutine when it was read into the system and remained fixed for all executions; program parameters were passed with the call to the subroutine. The programmer's task was to write a combining routine which would take the proper subroutines from tape and modify them appropriately. The motivation for these facilities was to reduce programming time; they were the first step in the direction of using a computer to help prepare programs for itself.

A second step was the use of "a programmer's language" instead of the machine's language. This evolved quite slowly. The use of symbols for machine operations came first, then relative numbers to which fixed relocation values could be added became common for instruction and data locations. However, by 1952 a "floating-address" notation had been developed by Wilkes [4, 5]: Any word or group of words could be designated by a letter or letter-number combination. The assembler for this language [5] had what was to become the classic structure. Two passes were made over the program: the first to assign a location to each word, and the second to fill in the correct values for the floating addresses.

Wilkes [4] also proposed "synthetic orders" which expanded into several parameterizable floating-address instructions. The expander acted as a preprocessor to the assembler, and its output could be saved to avoid unnecessary re-expansion on subsequent assemblies. Modern macro preprocessors are clearly elaborations of this early system.

The idea of a compiler had begun to develop and several "compilers" existed [6]. The programmer could write pseudo-codes to refer to subroutines, and the compiler performed the combining function envisioned by von Neumann. The function of a compiler was to interpret the pseudo-code, look up the subroutines in the library, then adjust and assemble them into a complete program. Although there is a vague resemblence between these early compilers and modern compilers, the structure and tasks of modern compilers had not yet emerged.

• *Early IBM assemblers*

The first symbolic assembly program produced by IBM was the NR9003 (described but not identified in [7]) for the 701. It was developed by the 701 engineering manager, N. Rochester, so that the engineers could test the machine. It made program modification easy by using symbolic addresses and by permitting the assembly of separately written programs. Different names for the same location could be associated by the use of a SYNonym pseudo-operation. Thus aliasing, the bane of compiler writers, made an early—albeit necessary—appearance.

The most widely used IBM assembly program for the 701 was the IBM S02 [8] written by William McClelland. This assembler supported relative addresses from an origin and performed extensive error checking. Hardware errors were frequent and often undetected by the hardware, so checking methods and restart procedures were important design criteria in early systems.

The concept of "regional programming" [8, 9] was used to effect program relocation: Storage was divided into regions into which the various programs, data, and, in at least one version of the idea [10, 11], dynamically overlaid, temporary storage for subroutines could be placed. This latter usage was very close to today's use of dynamic, stacked storage areas by procedures in such languages as PL/I. The implementation techniques were necessarily quite different since the 701 did not have index registers.

• *Early higher-level language processors*

IBM's first higher-level language was Speedcode [1, 12, 13] developed by John Backus in 1953 for the 701. He stated [12] that the ". . . most important reason for having a Speedcoding calculator, in addition to the 701, is a matter of economy . . . . Programming and testing costs often comprise between 50 and 70% of the total cost of operating a computing installation. . . . Speedcoding reduces coding and testing time considerably." The system was an interpreter which caused the 701 to behave like a three-address, floating-point calculator. Although it required over 30% of the memory (310 words), it supported an extensive repertoire of mathematical, I/O, and checking functions which were compactly expressed.

Indeed memory space, both primary and secondary, was a very scarce resource and strongly influenced the design of language processing systems for many years.

A compiling system, PACT I [14-20] for the 701 (and PACT IA [21] for the 704), emphasizing storage optimization, was started in late 1954 by a committee of IBM employees and customers. The compiler automatically allocated primary storage in regions for programs and data. In addition to space for the user's scalars, vectors, and arrays, special regions existed for temporaries and for the "perishable" data used by library routines. It was noted [19] that a compiler might be able to make an optimum automatic storage allocation but it would require that the compiler "follow the logic of the entire program." Steel noted [21] that ". . . logically it has a good deal of similarity to the celebrated 'four color' problem." With the exception of work by Ershov [22], these ideas were not fully explored until very recently [23, 24]. It is interesting to note that the compiler used a technique when constructing and accessing the table of variables which has since become known as hashing. This technique, in almost exactly the same form, is used commonly today. Knuth [25] cites H. P. Luhn with originating hashing and describing it in an IBM memorandum written in 1953. At about the same time another group of IBM people, Gene Amdahl, Elaine Boehm, Nathaniel Rochester, and Arthur Samuel, independently developed hashing for use in an assembly program they were writing for the 701. In 1979 a nice theoretical result [26] was obtained by J. L. Carter and M. Wegman which guarantees a good expected performance for any set of variables. It will undoubtedly become the standard hashing method used in future compilers.

• *SOAP: an optimizing assembler*
One of the earliest IBM systems to feature optimization was SOAP (Symbolic Optimizer and Assembly Program) for the IBM 650. By storing data and instructions on the 650 drum so that the drum was in the right position when data or the next instruction was required, a program might run as much as six or seven times faster [27]. While automatic optimization did not achieve such dramatic improvements, a factor of as much as 3.8 was obtained by a simple, two-pass preprocessor [28]. In designing this preprocessor, a predecessor of SOAP, optimum was taken as the point at which the improvement-per-unit-effort is maximized. The study of program optimization has continued to emphasize the development of faster techniques for improving the execution time of a program.

Most programs at the beginning of this early period were written in machine language. By the end of the period several automatic programming systems existed which provided a synthetic "computer" different from the real computer. Many of these systems were, however, costly to use, either because they were interpreters or because they relied on precoded routines to implement functions in the language not directly supported on the machine.

FORTRAN I

• *The project and its context*
Early in 1954 the FORTRAN I project was formed by John Backus. A fundamental question posed by the project was [29] ". . . can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?" A major goal [29] was to provide an automatic programming system which ". . . would produce programs almost as efficient as hand coded ones and do so on virtually every job." This seemingly impossible goal was met to an astonishing degree. In some cases it produced code which was so good that users thought it was wrong since it bore no obvious relationship to the source. It set a standard for object program efficiency which has rarely been equaled. The FORTRAN I compiler, begun in 1954 and completed in 1957, established modern compiler tasks, structure, and techniques. Indeed some of the techniques are still used in nearly the same form.

The compiler was developed for the 704, an IBM machine introduced in 1954 featuring built-in floating point and indexing capabilities. It compiled the FORTRAN I language which was defined as part of the project and evolved considerably as the project progressed. In order to achieve its efficiency goals, the high level arithmetic statements in the source program had to be translated so as to minimize storage references and, even more importantly, subscripts and their control had to make maximal use of the machine's three index registers. How all of this was achieved is described in [30], formalized in [31], and reviewed in [32] and [29]. This latter paper in particular contains a penetrating analysis of the project—its origins, development, and impacts—and should be read by everyone interested in computer science history or in compilers.

• *The overall organization of the compiler*
The compiler was divided into five sections (phases in today's terminology):

1. A statement identifier and arithmetic statement translator,
2. A subscript and DO statement analyzer,
3. A transformer which interfaced sections 2 and 4,
4. A control flow analyzer,
5. A global register allocator, and
6. Final assembly.

We now discuss sections 1, 2, 4, and 5 in more detail.

Translation
Today's compilers often use elegant, language-independent translator systems. The theory behind these systems did not really start to develop until the 1960s, but the problem appeared in its full form in this system. Given an arithmetic expression, the translator first created a sequence of arithmetic instructions, then transformed this sequence to eliminate redundant computations arising from the existence of common subexpressions (their term) and to reduce the number of accesses to memory. These transformations have been the subject of numerous investigations ([33] contains a good set of references), and we now know that an optimal solution is inherently hard.

Subscript and DO *statement optimization*
The translator did not complete the translation of DO statements and subscripts—that was the function of section 2, designed and developed by R. A. Nelson and I. Ziller. A symbolic index register corresponding to each particular subscript combination of a variable was created by the translator and existed until section 5 had assigned registers. The function of section 2 was to optimize the calculation of subscripts and DO control statements. The constant parts of the calculation were incorporated into operand addresses; operations involving DO control variables were transformed into index register increments when possible; loop-independent parts of the calculation were removed from the loop; and the loop exit test was transformed to use one of the registers needed for indexing. A nest of DO loops for array calculations was sometimes replaced by a single loop in the generated code! Some of these transformations are now subsumed in a somewhat more general optimization called "strength reduction" [34], but, with one possible exception [35], today's production compilers do not generally do as well.

Flow analysis
The function of sections 4 and 5 of the compiler was to assign real registers to the symbolic registers. Except for the symbolic registers and the assumption that they could all be assigned to real registers, the program on entry to section 4 was complete. The basic task, therefore, was to assign the symbolic registers to real registers so as to minimize the time spent loading and storing index registers. Section 4 of the compiler did a flow analysis of the program to determine the pattern and frequency of flow for use in section 5 where the actual assignment was made.

Basic blocks ("a basic block is a stretch of program which has a single entry point and a single exit point" [30]) were found and a table of immediate predecessor blocks constructed. Here, then, is the beginning of the elegant and fast control flow algorithms of today. Basic blocks and predecessor (successor) relationships are inputs to these algorithms.

The other task performed by section 4 was the computation of a probable frequency of execution of every predecessor edge. To do this Lois Haibt developed a Monte Carlo "execution" of the program with initial weights assigned to each edge. This method is no longer commonly used to identify frequently executed areas of a program; rather the program topology is used more directly but with less resultant precision.

Register assignment
Using the edge execution frequencies, regions were formed so that registers could be assigned to the most frequently executed areas (usually innermost loops), then to the next most frequently executed areas, etc., until the entire program had been treated. When a region had been processed, its entry and exit conditions were recorded: the values to be loaded on entry and stored on exit. A processed region was not re-examined when its containing region was processed, but the entry and exit conditions and whether or not it had any unassigned registers were used. The assignment of registers within a basic block used a "distance to next use" criterion to determine which register to displace when out of registers. "Activity bits" were used to determine the necessity of storing a value in a register for subsequent use if the register had to be reused. In case of register assignment mismatches across basic blocks, an attempt was made to permute the assignment.

This register assignment method, developed by Sheldon Best, was a phenomenal piece of work. The displacement algorithm for straight-line code was later proved optimal [36] for the "one-cost model" [37]: A displacement costs the same whether you need to store the register contents or not. Until 1980, when Gregory Chaitin [23] successfully applied a graph coloring algorithm to the global assignment of registers, most global assignment methods [38-40] were essentially variants on the FORTRAN I approach.

• *An assessment*
Perhaps the best way of demonstrating the results of this project is to show an example of its output—an output which startled this author. The FORTRAN program in Fig. 1 moves array B to array A. The assembly program shows this being done with one loop instead of the two expected from the source. (FORTRAN stored its arrays column-wise and backwards; the 704 subtracted the value in the index register from the address.)

The real results of the project are the influences it had on future compilers and theory. Some of these effects have already been mentioned; more will be discussed later. Suffice it to say that the technological fallout from this project has been extensive.

The late fifties and early sixties

• *Characteristics of the period*

The period from 1957, when the 709 was announced, to 1964, when System 360 was announced, was one of great optimism and little discipline. New languages, computers, systems, and ideas appeared at an astonishing rate. By 1964 IBM implementations of FORTRAN, Comtran, COBOL, RPG, Autocoder, PRINT, FORMAC, GPSS, and other languages existed on at least one of the computers introduced during the period: the 709-90, 707-70, 1401, 1410-7010, Stretch, 7040-44, and others. Total systems, such as IBSYS for the 7090, provided a unified execution environment and a central set of facilities to exploit the capabilities of such hardware features as buffered I/O. Time-sharing systems, such as Quiktran, provided on-line, interactive facilities. Formalisms emerged, and the understanding and use of fundamental data structures such as lists and trees became widespread. However, the spirit of the period is probably best exemplified by the interest in universal compiling systems. The obvious problem created by the proliferation of languages and computers led to numerous efforts to provide a single system capable of compiling multiple source languages for multiple target machines.

• *Some interesting systems*

Language processors for the 709, 7090-94 systems

The first system designed for the 709, 7090-94 computers was SOS or SCAT (SHARE Compiler Assembler Translator) [41–46]. It was a very complex system designed by a committee spread throughout the country and representing many different companies. It was to provide source-level debugging without the cost of complete retranslation whenever the program changed or additional information was needed. The central mechanism was a "SQUOZE" form of programs arising from the partial translation of various source inputs, including incremental changes from the user. It never became a production system. It was late, the general quality was poor, and there were serious performance problems in "loading" multiple modules of SQUOZE decks. Furthermore, there was a reluctance to change from the existing patterns of program development and execution.

```
                 SOURCE

         DIMENSION A(10,10)
         DIMENSION B(10,10)
         . . . . .
         DO 1 J=1,10
         DO 1 I=1,10
       1 A(I, J)=B(I, J)
         . . . . .

                 RESULT

        LXD ONE, 1          load 1 into regl
LOOP    CLA B+1,1
        STO A+1,1
        TXI *+1,1,1         add 1 to regl and
                            goto next inst
        TXL LOOP,1,100      if regl≤100
                            goto loop
        . . .
ONE        ,,1              data value one
A       BES 100             reserve 100 locs,
                            ending with A
B       BES 100             reserve 100 locs,
                            ending with B
```

Figure 1 FORTRAN I translation of array move.

Concern for the compile-time cost of translating programs did not, however, disappear. In fact the original FORTRAN I system had already been modified to handle FORTRAN II—a language featuring subroutines and COMMON blocks of shared variables. These extensions together with the BSS loader were a significant factor in the acceptance and effective use of FORTRAN for large programming applications. Assembly language programs and separately compiled programs could be loaded and executed as a unit.

In 1961 a decision was made to completely rewrite the FORTRAN compiler to improve the compile speed and to support the new FORTRAN IV language. Another factor in the rewrite decision was the new, integrated system, IBSYS, for the 7090: The new FORTRAN would be an integrated component. The new compiler [47] was neither very fast nor did it produce object code as efficient as the original compiler. The new compiler did, however, introduce a new transformation: "anchor pointing" to optimize FORTRAN IV's logical expressions. The evaluation order of the components of a logical expression was revised to minimize the time required for evaluation.

The Comtran and COBOL compilers

A compiler for Comtran (Commercial Translator), an early IBM language supplanted by COBOL, was delivered in early 1961. The Comtran compiler contained a number of new and very innovative techniques. Unfortunately, they were never published. The project manager, Richard Talmadge, invented a table driven scanner (probably the first) and extended the PACT IV hashing methods for use in this system. Robert Rock developed a nice method for handling *n*-tuples. Much of the system design was used in the 7090 COBOL, where it was augmented by a very early table management system, CITRUS, written by Claude Coray.

The scanner used an operator precedence table with a mechanism for handling parentheses and varying numbers of operands. As the input string was scanned, a table of operators was built giving the precedence and position of each. This table was then sorted by precedence value and used to generate *n*-tuples from the text string. Each unique *n*-tuple was assigned a unique symbolic register to hold the result, and the text string was dynamically modified (reduced) to the symbolic register replacing the operator and operands. Before assigning a symbolic register to an *n*-tuple, a search was made to determine if it had occurred previously in the statement. If it had, the *n*-tuple was not put out—a common subexpression had been found. On completion of the scan of a statement the number of symbolic registers used in the output was minimized.

The scanner was driven by a table containing a set of syntax vectors. Each vector contained an encoded question related to the scan state and two branch target locations: one if the answer was true, the other if false. An interpreter used the table to determine the routing to the target action routines.

The dictionary organization was strikingly similar to a form commonly used today. The source program names were stored in a symbolic dictionary, attributes in an attribute table. These tables were packed, and hashing through a sparse intermediate table was used to look up names.

These techniques were carried over to the COBOL compiler developed for IBSYS [48]. In addition to incorporating these early and excellent techniques, the 7090 COBOL compiler was also one of the first systems [49] to use a dynamic table management and spill system: CITRUS (for Coalesced Indirect Table Reference Unification Scheme). The compiler's tables could be opened, closed, relocated, expanded, and contracted during execution.

FORTRAN for the 1401
The 1401, announced in 1959, was a small, character-oriented computer imposing a severe set of constraints on the FORTRAN compiler design [50]. It was designed to operate in 8000 storage locations (characters) with tape usage being optional. We mention the FORTRAN compiler for the 1401 in this history because it exemplifies an interesting design for a small system. The compiler consisted of 64 phases, the first three of which were loaded before the program. The entire program was then read in, and the remainder of the compiler was passed against the program, one phase at a time. When the compiler was finished, the program was in executable form in memory and could be executed immediately. An object deck was optionally available. While the system did not permit arbitrarily large programs, it made very effective use of the space available and did not incur the overhead associated with storing intermediate results on tape.

Quiktran
In 1963, Quiktran, IBM's first time-sharing system, was operational [51, 52]. Developed by J. Morrissey, T. Dunn, J. Keller (Rivlin), E. Strum, and G. Yang, it supported 40 concurrent users on the 7040-44 computers with a FORTRAN interpretive system providing interactive, source-language-level debugging. The system could act as a desk calculator and could dynamically alter the program during execution. Each executable statement was stored internally as a polish string created by using a bi-directional "forcing table," which was also used to recreate the source. Because of this capability and because the language was standard FORTRAN, a program could be debugged using this system, then recreated for compilation by a standard FORTRAN compiler. The PL/I Checker [53], currently available for debugging PL/I programs, provides many of these same facilities but does not recreate the source from the internal form used by the interpreter.

The Stretch-Harvest compiler
An early and persistent goal of compiler designers has been the construction of a single compiler for multiple source languages and multiple target machines. The compiler for the Stretch-Harvest computers compiled FORTRAN and Alpha language programs and was designed to handle other languages as well. The two source languages were as dissimilar as FORTRAN and COBOL, and the Harvest attachment to Stretch bore no resemblance to its host. One of the central notions in the design of many such systems is that a common internal (intermediate) language (IL) can be found for expressing all the source languages. Ideally this common IL is such that very little language-specific code is needed in the system. In the Stretch-Harvest compiler the common IL was a high level Autocoder to which each source language could be translated by one pass.

Having translated the source to IL, the second phase of the compiler made three passes over the program to map storage and expand subscripts, to collect flow and symbolic register-usage patterns, and to assign index registers. The second phase produced macros for input to a subsequent macro-assembler. A variant of an early list processing system [54] developed by H. Gelernter, J. Hansen, and C. Gerberich was used in phase 2 to manage the internal data structures.

The system was not a success; compilation time was unacceptable. The interfaces between phases were in externally readable form, costly to construct and costly to read and write; either language, and FORTRAN in particular, could have been compiled much more efficiently by a language-specific system. Compiler technology, and the art of effectively utilizing it, was not advanced enough in 1960 for such a system.

- *Universal compiling systems*

The Stretch-Harvest compiler design was, however, conservative when compared with proposals for some other language processing systems. Motivated by the proliferation of languages and computers and by the continued scarcity of skilled programmers, particularly experienced compiler designers, the idea of a universal compiling system became very popular. In the late 1950s there was considerable interest in the design of an UNCOL (UNiversal Computer Oriented Language) [55] to which any high level language could be translated and from which executable programs could be created for any machine. A new language would require a new translator to UNCOL, and then compilers for that language would exist on all supported computers. Similarly a new computer required one translator from UNCOL to get a family of compilers. Though investigators soon despaired of finding such a language, the motivating factors still existed.

Within IBM there were two universal compiling systems projects in the early 1960s: XTRAN [56] and SLANG [57]. Julian Green's XTRAN approach was to provide a boot-strapped, list-processing system which transformed the source string to the object string. It used auxiliary information about source symbols and operators to translate the source language to a macro language, then used macro definitions to translate the macro language to machine language, and finally optimization information to optimize the machine language. This approach was intended to minimize the consequences of changes to the processor algorithm, the source language, or the object language. In 1962 a decision was made to terminate the XTRAN project and concentrate on the much more ambitious SLANG (Systems Language) project being developed by R. Sibley.

SLANG was both a problem-oriented, machine-independent language and a compiler with a machine description capability. The main emphasis in its initial formulation was to achieve machine independence, thereby providing portability of compilers written in SLANG and, since the SLANG compiler was written in SLANG, of the SLANG system itself. As the system developed, a syntax-directed facility was added to provide source language independence through the use of an augmented BNF (Backus Normal Form) [58] description of the source language being compiled. The source language description was compiled and used to translate the input program to EMILs—Elementary Machine Intermediate Language statements. Tables defining the target machine characteristics were then used to translate EMILs to object code.

The SLANG project was terminated in 1965 when it became apparent that the object code produced, and hence the SLANG system and all compilers written in it, was too inefficient. Machine independence failed for several reasons: there was no adequate way to parameterize the description of the machine—particularly its I/O; when writing a program in SLANG, users generally optimized it for a particular machine; and the effect of storage size on the overall organization of a compiler could not be described. Language independence failed due to the lack of a satisfactory method of semantics specification and because the parameterization was done with respect to existing languages (FORTRAN and COBOL). Any language with new or extended features became a problem to describe. In particular the new language, PL/I, was too difficult for SLANG.

It is now generally accepted that a special compiler-writing language is not necessary, but if the goal is a reasonably efficient compiler written in its own language, it is essential that that compiler be capable of producing good code. Techniques for providing both a machine descriptive capability and producing good code are only now beginning to emerge.

- *Early developments in the theory of parsing*

Program translators today usually use a lexical analyzer (scanner) and a syntax analyzer (parser) to recognize the substrings in the source string. Lexical analyzers recognize the tokens (operators, separators, identifiers, etc.) using transition diagrams and finite state automata. Syntax analyzers recognize the substrings (phrases) using a context-free grammar. In fact efficient parsers can be automatically constructed from a grammar. Many of these elegant systems have their roots in the practical and theoretical efforts of this period.

The syntactic notation BNF, commonly used to specify a grammar, was introduced by John Backus in 1959 [58]. It is discussed elsewhere in this issue.

A major contribution to the theory of parsing was made by John Cocke. Though not fully documented at the time, it was mentioned in [59], and a variant, the nodal spanning parse, appeared in 1969 [60]. An excellent discussion of the relevance of Cocke's parsing method was given by Robert Floyd in his 1978 Turing Award Lecture [61] and is quoted in the next paragraph.

"In the early 1960's, parsing of context-free languages was a problem of pressing importance in both compiler development and natural linguistics. Published algorithms were usually both slow and incorrect. John Cocke, allegedly with very little effort, found a fast and simple algorithm [62], based on a now standard paradigm which is the computational form of dynamic programming [63]. The dynamic programming paradigm solves a problem for given input by first iteratively solving it for all smaller inputs. Cocke's algorithm successively found all parsings of all substrings of the input. In this conceptual frame, the problem became nearly trivial. The resulting algorithm was the first to uniformly run in polynomial time."

The mid- and late sixties

• *Characteristics*

On April 7, 1964, IBM announced the System/360. The announcement included five basic, compatible computer models with 19 combinations of speed and storage capacity; the largest processor was 100 times more powerful than the smallest, and the main storage capacity ranged from 8192 bytes to more than eight million. The system was supported by a powerful operating system incorporating several data base access methods and interfaces to numerous auxiliary storage and input-output devices.

Language processors for FORTRAN, COBOL, the new PL/I language, and other languages had to support most of the configurations and to function in the environment supplied by the operating system. These requirements, together with the fact that many design decisions were being made in parallel, created a rather formidable challenge for the designers of these systems. Several "design points" determined by the maximum memory size available to a system were established to cope with the memory constraints. Families of upwards-compatible language processors were then produced. For example, three distinct FORTRAN compilers were written for use with the main operating system OS: an E level requiring 32 000 bytes, a G requiring 128 000, and an H requiring 256 000.

Most of the language processor efforts during this period can be characterized as the engineering of existing technologies to fit the constraints. FORTRAN H was an exception. We now discuss that work and some other technological and scientific advances which appeared in the late 1960s.

• *FORTRAN H*

C. W. Medlock and E. Lowry, designers of the FORTRAN H optimizer, stated [38] that: "For small loops of a few statements (the FORTRAN H compiler) very often produces perfect code." This was accomplished by global (to a subroutine) optimizations which generalized and thus extended the transformations used in earlier FORTRAN compilers, particularly FORTRAN I. Control flow analysis of the program was based, as in FORTRAN I, on a control flow graph in which the nodes represented basic blocks. Loops arising from constructs other than DOs were recognized, and "back dominators" were found. A back dominator of a node represents code which has to be executed before code in the node of interest is executed, and thus is a natural place to look for common subexpressions and to place code when the back dominator is less frequently executed. All subexpressions, not just those related to address calculations arising from subscript expansions, were considered—one subexpression at a time. Advances in the technology since that time permit the identification of all formally identical common subexpressions in parallel. Numerous other transformations and analyses, including a limited form of data flow analysis, were incorporated.

The efficacy of these optimizations is demonstrated by the fact that fifteen years after its release, it is still one of the best product-compilers in terms of object code efficiency. (Some improvements have been made by R. G. Scarborough, primarily in the area of subscript expansions and register allocation [35].) The design was incorporated into an optimizing compiler for PL/I (PLIOPT) and also proved adequate for supporting the FORTRAN compiler itself, 70% of which was written in FORTRAN.

• *PL/I*

Although the initial plans for implementing the newly defined PL/I language for System 360 included a family of compilers, only two, PL/I F and PL/I D, were produced. As it was, this was a monumental effort requiring great ingenuity: The language was large, the design point small, the operating system interactions extensive. The implementation of the new and comprehensive facilities for storage management, interrupt handling, and multitasking were particularly innovative. J. Cox, J. Nash, and N. Clarke were most responsible for pulling off the enormous engineering effort which at its peak involved some 70 people.

Subsequent to the release of the PL/I F compiler, a technique for optimizing PL/I was investigated [64] by M. Elson, R. Larner, S. Rake, and others. The thesis of the investigation was that context sensitive ("special casing") code generation techniques were needed to produce good code for a language as rich as PL/I on a machine such as the 360 with its multiplicity of possible code sequences for even the simplest of functions. The form of the investigation was to produce a prototype system in which each statement was stored as a tree augmented with the data attributes, and a tree-walking, interpretive subsystem was used to generate code. Although this investigation did not directly result in a product, the thesis has remained attractive. The problem with it is the size of the code generators and the consequent probability of errors. Furthermore, it cannot deal effectively with transformations requiring more global information, such as register assignment across loops. A recent system, based on a diametrically opposite thesis, is discussed later.

• *APL*

Interpreting code is often at least an order of magnitude slower than executing it directly. The APL system is a counter example. L. Breed and R. Lathwell, who, together with Roger Moore, built the system, state [65] that: "The APL processor is interpretive; however, because of the efficiencies afforded by array operations, program execution is often one-tenth to one-fifth as fast as compiled code." Another factor is the excellence of the design. The system, initially implemented on the 7090 and then, in 1966, on the 360, consisted of a supervisor and the interpreter. It was designed as a total system in which the supervisor had complete control over system resources and the supervisor-interpreter communications could be simplified. The anticipated use of the system and the language constructs were carefully considered when making design choices. An interesting evaluation of these choices and the historical setting in which they were made is given in [66].

• *The ACS project*

The Advanced Computing Systems (ACS) project was a hardware-software project started in 1964 as System Y at the IBM Thomas J. Watson Research Center. The aim of the project was the design of a very high performance system for the large scientific market. From the beginning of the project it was recognized that the only way to realistically realize the performance goals and make them accessible to the user was to design the compiler and the computer at the same time. In this way features would not be put in the hardware which the software could not use or which the software could support more effectively. The CPU, for example, had a high degree of parallelism, both in fetching instructions and data and in instruction execution. Could the compiler schedule the instruction stream to take advantage of the parallelism? It turned out in fact that the experimental optimizing compiler developed to evaluate CPU designs could sometimes do better than carefully hand-optimized code.

In order to isolate the effects of changes in the CPU design and to modularize the experiment, the ACS compiler classified the optimizations as machine independent or machine dependent. The machine independent analyses and optimizations [67] included a general control flow analyzer, common subexpression elimination and code motion, data flow analysis, strength reduction, constant propagation, and dead code elimination; the machine dependent transformations included scheduling [68] and register allocation [39, 69]. With the exception of scheduling, most of the other analyses and transformations had appeared in previous IBM compilers, particularly FORTRAN I and FORTRAN H. In the ACS compiler (which, like the entire ACS system, never became a product) the techniques were generalized and isolated as much as possible from source and machine language constructs. This is an essential step, of course, in the evolution of general compiler building tools. Out of this project came, therefore, numerous advances in compiler technology and, more importantly, the foundations of the theory of program analysis and optimization. The elaboration and refinement of this theory was the major development in language processor technology during the next decade.

Recent history: the seventies

• *Characteristics*

Increasing maturity is perhaps the hallmark of computer science and technology in the 1970s. The cycle from pragmatic problem (with a heuristic solution) to formal solution to realization of the solution in a practical system has been completed, or at least well started in numerous cases. The study of algorithms has led to a growing collection of tools whose time and space bounds and correctness are known. Programming has become more disciplined, and formal mechanisms for describing and verifying various properties of systems have been extensively investigated.

Language processor technology has been both the chief benefactor and the *raison d'etre* for much of the work. Parsing and program optimization are the quintessential examples of the evolution of the technology.

At the beginning of this decade, the theory of parsing was well understood; by the end of the decade, a fast, reliable parsing system is one of the "on-the-shelf" tools a programmer uses when building a compiler. At the beginning of the decade, the theory of program analysis and optimization was in its infancy; now some of the results of these investigations are appearing in various implementations. Register allocation is another area in which both theoretical and practical advances have been made. In this section we trace the evolution of these latter technologies and briefly describe one implementation utilizing them.

- *Program analysis and optimization*

In 1970 two papers [70, 71] described fast, abstract methods for program control and data flow analysis. Given a directed graph representation of a program, one paper [70] showed how the graph could be partitioned into "intervals" (single entry subgraphs in which all loops contain the entry node) and, treating each interval as a node, higher order graphs were derived and then analyzed to effectively codify the nesting structure of the program. The algorithm was abstract in that it did not depend on specific source language constructs, such as DOs, and it was fast when applied to typical program graphs, though it had a poor worst-case bound. For each graph it was linear in the number of edges, and, since the number of derived graphs rarely exceeds three in practice (the depth of nesting of loops), the number of derived graphs was small. Since that time a nearly linear algorithm has been found [72] which also gives a somewhat better codification of interesting control flow relationships. However, the most important feature of the interval analysis method was that a node ordering was established which could be used to rapidly determine other relationships in the program.

The second paper [71] dealt with a particularly interesting application: common subexpression elimination and code motion. All redundant, formally identical (look-alike) subexpressions in the entire program could be identified by one execution of a fast algorithm—having the same time bound as the interval analyzer. The algorithm not only recognized subexpressions which could be eliminated from the program when the computation already existed on all paths to the subexpression, but it also found code which could be moved out of a loop or nests of loops.

Another important application was finding "def-use" relationships: Given a definition of a variable, all potential uses of that definition are of interest during optimization and register allocation. The relationship and its dual, the use-def relationship, are needed for strength reduction [34], constant propagation, dead code elimination, and other transformations [73]. The interval-based algorithm [74] for finding all such relationships was fast in the sense described previously; now nearly linear algorithms [75] exist.

A natural consequence of the existence of fast analysis and optimization algorithms for individual procedures, as well as the development and maintenance of on-line program data bases and advances in programming methodologies, was an interest in analyzing and transforming collections of procedures. The first investigations [76, 77] into interprocedural analysis defined the problem and outlined pragmatic solutions. Elegant, though not necessarily pragmatic or completely general, algorithms [78-80] now exist. A research project, the Experimental Compiling System [81-83], is investigating the use of interprocedural analysis [83, 84] and optimization in a compiler building system. The basic ideas are to use procedures to express the semantics of the language being compiled, to deduce the characteristics of the language by analysis, to use procedure integration (in-line expansion) to do code generation, and to optimize in order to customize and obtain good code.

The work on program analysis and optimization has many parallels with the work on parsing and will undoubtedly prove to be as important. It seems almost certain that language-independent tools for analyzing and optimizing programs will become as available as lexical analyzers and parser-generators are now. Also, just as our understanding of parsing has affected language design, so will our understanding of optimization. It seems unlikely that new languages will continue to contain constructs such as unconstrained aliasing which confound analyses (as well as users).

- *Global register allocation*

One of the very difficult functions of an optimizing compiler is register allocation. It has been formulated as an integer programming problem [85] and shown to be a hard problem [86] even for straight-line code. Practical implementations are necessarily heuristic and typically complex. Recently a fast (usually) and surprisingly good (approaching that of hand-coded assembly language) approach has been developed and implemented [23] by G. Chaitin. The problem is formulated as a graph coloring problem: Each node in the graph stands for a computed quantity that resides in a machine register, and two nodes are connected by an edge if the quantities interfere with each other, that is, if they are simultaneously live at some point in the object program. The problem is to assign different colors (registers) to connected nodes. Obtaining an optimum coloring is hard, but the implementation showed that a fast heuristic method for assigning colors to these particular graphs generally resulted in a very good assignment. Another interesting aspect of the work was that most of the idiosyncrasies of the target machine (*e.g.*, register pairs, dedicated registers) could be handled uniformly and systematically. It seems likely that this approach to register allocation will be the basis for many interesting investigations and elegant implementations.

• *An implementation*
In our discussion of PL/I implementations in the preceding section, we said that context-sensitive (special casing) code generation was often thought necessary to obtain good code for complex languages. In an experimental compiler [87] for a variant of PL/I a different approach was used. Code is produced when some grammatical construct is recognized by a LALR (Look Ahead Linear Right)-produced parser-generator and is not, therefore, selected because of its context. This simple and straightforward code is then optimized by a series of stand-alone programs implementing the mathematically based algorithms mentioned earlier. Code selection tricks are avoided. Registers are then assigned by graph coloring. The method, which is largely language and machine independent, can result in final code approaching and even surpassing hand-coded assembly-language programs. This result by itself is not surprising—the FORTRAN I compiler accomplished the same thing; its importance lies in the means by which it was obtained.

The uniform, systematic use of mathematically based, general algorithms in a compiling system leads to a simpler, more predictable design and a more maintainable implementation. This is widely accepted. What we now know is that this method can also produce better code.

Summary

We have traced the history of language processor technology in IBM from its rather slow start in 1952 to the current collection of elegant algorithms, powerful techniques, and unsolved problems. General, mathematically based algorithms now exist for parsing, analyzing, and optimizing programs. IBM has made significant contributions to these areas, particularly the last two. The art of designing and constructing a language processor system has become more scientific and, in some cases, quite routine. Assemblers, interpreters, and macro systems are generally well understood, though they still require the judicious selection of techniques to fit the requirements. Though worked on in one form or another for more than twenty years, a general production-quality compiler building system does not exist. We are, however, much closer to this goal: We now have language and machine independent translators; recent experiments with similarly general analyzers, optimizers, and register allocators increase the likelihood of such a system being developed.

A constant motivating force for work in language processors has been "the programming problem"—ours and our customers'. Providing effective systems for languages (Speedcode, FORTRAN, and APL are prime examples) has permitted and encouraged users to concentrate more on problem solving and less on program writing. Providing increasingly more general language processor tools has permitted the construction of more diverse and reliable systems.

The synergistic relationship between theory and practice, noted throughout the paper, leads to two observations: All (or nearly all) theoretical results embodied in practical implementations concern problems initially identified in such implementations; all (or nearly all) powerful general tools utilize theoretical results.

Some final observations

"Programming is optimization" [88]. Most if not all of the choices made in programming a solution to a computable problem are aimed at achieving an acceptable level of efficiency. For example: How is information to be represented? Should it be sorted to permit fast lookups? What sort method should be used? These are optimization questions, not problem solving questions. If programming is to truly become problem solving on a computer, we must relieve users of such decisions by providing the technology to permit widespread use of very high level, problem-oriented languages and tools. In his paper on the history of FORTRAN [29], John Backus corroborates this view. The next paragraph is from that paper.

"To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed. In fact, I believe that we are in a similar, but unrecognized, situation today: in spite of all the fuss that has been made over myriad language details, current conventional languages are still very weak programming aids, and far more powerful languages would be in use today if anyone had found a way to make them run with adequate efficiency. In other words, the next revolution in programming will take place only when *both* of the following requirements have been met: (a) a new kind of programming language, far more powerful than those of today, has been developed and (b) a technique has been found for executing its programs at not much greater cost than that of today's programs."

I believe much of the technology is in place to support this revolution. The next decade could be very exciting as existing technologies are exploited and new ones developed to subsume much more of the programmer's task.

Acknowledgments

In the paper I have indicated a few of the IBM people who contributed to the development of language processor technology in IBM. Strong language processor groups at

IBM's Time-Life Center in New York City (with M. Ackroyd, G. Garabedian, W. Heising, R. Larner, L. Levine, F. Pessin, P. Smith, and B. Weitzenhoffer to name a few), at Hursley in England (with B. Marks, J. Nash, and I. M. Clarke), and at numerous other locations contributed most to the solid development of the area. Many others, including the fine scientists and practitioners outside IBM with whom we have collaborated and whose work we have admired and assimilated, must unfortunately remain unrecognized here.

Two people, John Backus and John Cocke, deserve special re-acknowledgment for their many major contributions to language processor technology. John Backus developed Speedcode, made FORTRAN happen (with the help of a very fine group [30]), and invented BNF among other things. John Cocke has contributed to nearly every aspect of language processor science and technology, program analysis and optimization, parsing, register allocation, and macro systems, as well as to areas such as improved hardware architecture which make these tasks easier.

These are, as we said, just a few of the people who made the history of language processing. I also want to acknowledge a few of the people who helped with this paper: C. Alberga, J. Cox, M. Hopkins, J. Palmer, and B. Weitzenhoffer supplied invaluable memoranda, manuals, and names, and the Yorktown Research librarians, I. Cawley, E. Howing, and J. Leonard, cheerfully helped locate many obscure documents.

In 1962 Donald Knuth said [89] that the early history of compilers was difficult to assess. It has not become any easier but, though the assessment is mine, the resources available to me have contributed immeasurably to the pleasure of the task.

References

1. John W. Backus and Harlan Herrick, "IBM 701 Speedcoding and Other Automatic-Programming Systems," *Symposium on Automatic Programming for Digital Computers*, Office of Naval Research, May 1954, pp. 106–113.
2. H. H. Goldstine and J. von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," The Institute for Advanced Study, Princeton, NJ, 1947.
3. Maurice V. Wilkes, David J. Wheeler, and Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Publishing Company, Reading, MA, 1951.
4. M. V. Wilkes, "Pure and Applied Programming," *Proceedings of the ACM Meeting at Toronto*, September 1952, pp. 121–124.
5. Charles W. Adams, "Small Problems on Large Computers," *Proceedings of the ACM Meeting in Pittsburgh*, May 1952, pp. 99–102.
6. Richard K. Ridgeway, "Compiling Routines," *Proceedings of the ACM Meeting at Toronto*, September 1952, pp. 1-3.
7. Nathaniel Rochester, "Symbolic Programming," *IRE Trans. Electron. Computers* **EC-2,** 10–15 (1953).
8. "Notes on Programming Systems and Techniques for the IBM Type 701 Electronic Data Processing Machines," Prepared for the Applied Science Course held during the week beginning March 16, 1953.
9. "Regional Programming," Notes (from N. Rochester) for 701 customer class held August 25 to August 28, 1952.
10. C. L. Baker, "Symbolic Coding for Type 701 Calculator," Douglas Aircraft Company, Inc., Santa Monica, CA, February 1953.
11. "Notes on the 701 Symposium held at Douglas Aircraft Company, Santa Monica, California," August 1953.
12. J. W. Backus, "The IBM 701 Speedcoding System," *J. ACM* **1,** 4-6 (1954).
13. *IBM Speedcoding System for the Type 701*, IBM Corporation, September 1953.
14. Wesley S. Melahn, "A Description of a Cooperative Venture in the Production of an Automatic Coding System," *J. ACM* **3,** 266-271 (1956).
15. Charles L. Baker, "The PACT I Coding System for the IBM Type 701," *J. ACM* **3,** 272–278 (1956).
16. Owen R. Mock, "Logical Organization of the PACT I Compiler," *J. ACM* **3,** 279-287 (1956).
17. Robert C. Miller, Jr., and Bruce Oldfield, "Producing Computer Instructions for the PACT I Compiler," *J. ACM* **3,** 288–291 (1956).
18. Gus Hempstead and Jules I. Schwartz, "PACT Loop Expansion," *J. ACM* **3,** 292-298 (1956).
19. J. I. Derr and R. C. Luke, "Semi-Automatic Allocation of Data Storage on PACT I," *J. ACM* **3,** 299-308 (1956).
20. I. D. Greenwald and H. G. Martin, "Conclusions After Using the PACT I Advanced Coding Technique," *J. ACM* **3,** 309-313 (1956).
21. T. B. Steel, Jr., "Pact IA," *J. ACM* **4,** 8-11 (1957).
22. A. P. Yershov (also Ershov), *The Alpha Automatic Programming System*, Academic Press Ltd., London, 1971.
23. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein, "Register Allocation via Coloring," *Computer Languages* **6,** 47-57 (1981).
24. Janet Fabri, "Automatic Storage Optimization," *Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices* **14,** 83-91 (August 1979).
25. Donald E. Knuth, *The Art of Computer Programming, Volume 3: Searching and Sorting*, Addison-Wesley Publishing Company, Reading, MA, 1973.
26. J. Lawrence Carter and Mark N. Wegman, "Universal Classes of Hash Functions," *J. Computer Syst. Sci.* **18,** 143-154 (1979).
27. G. R. Trimble, Jr., and E. C. Kubie, "Principles of Optimum Programming of the IBM Type 650," *IBM Applied Science Division Technical Newsletter No. 8*, 1954.
28. Barry Gordon, "An Optimizing Program for the IBM 650," *J. ACM* **3,** 3-5 (1956).
29. John Backus, "The History of FORTRAN I, II, and III," *ACM SIGPLAN History of Programming Languages Conference*, *SIGPLAN Notices* **13,** 165–180 (August 1978).
30. J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The FORTRAN Automatic Coding System," *Proceedings Western Joint Computer Conf.*, Los Angeles, 1957, pp. 188-198.
31. Peter B. Sheridan, "The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System," *Commun. ACM* **2,** 9-21 (1959).
32. J. W. Backus and W. P. Heising, "FORTRAN," *IEEE Trans. Electron. Computers* **EC-13,** 382-385 (1964).
33. Alfred V. Aho, "Translator Writing Systems: Where Do They Stand?", *Computer* **13,** 9-14 (1980).

34. F. E. Allen, J. Cocke, and K. Kennedy, "Reduction of Operator Strength," *Data Flow Analysis*, Neil D. Jones and Steven S. Muchnick, Eds., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981, pp. 79-101.
35. Randolph G. Scarborough and Harwood G. Kolsky, "Improved Optimization of FORTRAN Object Programs," *IBM J. Res. Develop.* **24**, 660-676 (1980).
36. L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Syst. J.* **5**, 78-101 (1966).
37. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, "Index Register Allocation," *J. ACM* **13**, 43-61 (1966).
38. Edward S. Lowry and C. W. Medlock, "Object Code Optimization," *Commun. ACM* **12**, 13-22 (1969).
39. J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM J. Res. Develop.* **18**, 20-39 (1974).
40. William Harrison, "A Class of Register Allocation Algorithms," *Research Report RC 5342*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.
41. Donald L. Shell, "The SHARE 709 System: A Cooperative Effort," *J. ACM* **6**, 123-127 (1959).
42. Irwin D. Greenwald and Maureen Kane, "The SHARE 709 System: Programming and Modification," *J. ACM* **6**, 128-133 (1959).
43. E. M. Boehm and T. B. Steel, Jr., "The SHARE 709 System: Machine Implementation of Symbolic Programming," *J. ACM* **6**, 134-140 (1959).
44. Vincent J. DiGri and Jane E. King, "The SHARE 709 System: Input-Output Translation," *J. ACM* **6**, 141-144 (1959).
45. Owen Mock and Charles J. Swift, "The SHARE 709 System: Programmed Input-Output Buffering," *J. ACM* **6**, 145-151 (1959).
46. Harvey Bratman and Ira V. Boldt, Jr., "The SHARE 709 System: Supervisory Control," *J. ACM* **6**, 152-155 (1959).
47. R. Larner, "Design of an Integrated Programming and Operating System Part IV: The System's FORTRAN Compiler," *IBM Syst. J.* **2**, 311-321 (1963).
48. R. T. Dorrance, "Design of an Integrated Programming and Operating System Part V: The System's COBOL Compiler," *IBM Syst. J.* **2**, 322-327 (1963).
49. Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1969.
50. L. H. Haines, "Serial Compilation and the 1401 FORTRAN Compiler," *IBM Syst. J.* **4**, 73-80 (1965).
51. T. M. Dunn and J. H. Morrissey, "Remote Computing—An Experimental System: Part 1: External Specifications," *AFIPS Conf. Proc.* **25**, 413-423 (1964).
52. J. M. Keller, E. C. Strum, and G. H. Yang, "Remote Computing—An Experimental System: Part 2: Internal Design," *AFIPS Conf. Proc.* **25**, 425-443 (1964).
53. B. L. Marks, "Design of a Checkout Compiler," *IBM Syst. J.* **12**, 315-327 (1973).
54. H. Gelernter, J. R. Hansen, and C. L. Gerberich, "A FORTRAN-Compiled List-Processing Language," *J. ACM* **7**, 87-101 (1960).
55. J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The Problem of Programming Communication with Changing Machines: A Proposed Solution," *Commun. ACM* **1**, 12-18 (1958) and *Commun. ACM* **1**, 9-16 (1958).
56. R. W. Bemer, "Survey of Modern Programming Techniques," *The Computer Bulletin*, 127-135 (1961).
57. R. A. Sibley, "The SLANG System," *Commun. ACM* **4**, 75-84 (1961).
58. J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proceedings International Conference on Information Processing, UNESCO, Paris*, Butterworths, London, 1960, pp. 125-132.
59. David G. Hays, "Automatic Language-Data Processing," *Computer Applications in the Behavioral Sciences*, Harold Borko, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1962.
60. John Cocke and J. T. Schwartz, "Programming Languages and Their Compilers," Courant Institute of Mathematical Sciences, New York University, New York, 1969.
61. Robert W. Floyd, "The Paradigms of Programming," *Commun. ACM* **22**, 455-460 (1979).
62. Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.
63. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1974.
64. M. Elson and S. T. Rake, "Code-Generation Technique for Large-Language Compilers," *IBM Syst. J.* **9**, 166-188 (1970). (Republished in *Compiler Techniques*, Barry W. Pollack, Ed., Auerbach Publishers, Inc., Pennsauken, NJ, 1972, pp. 410-438.)
65. L. M. Breed and R. H. Lathwell, "The Implementation of APL/360," *Interactive Systems for Experimental Applied Mathematics*, Melvin Klerer and Juris Reinfelds, Eds. (Proceedings of the ACM Symposium held in Washington, DC, August 1967), Academic Press, Inc., New York, 1968, pp. 390-399.
66. E. E. McDonnell, "The Socio-technical Beginnings of APL," *ACM SIGPLAN/STAPL* **10**, 13-18 (December 1979).
67. F. E. Allen, "Program Optimization," *Annual Review in Automatic Programming*, Vol. 5, Mark I. Halpern and Christopher J. Shaw, Eds., Pergamon Press, Inc., Elmsford, NY, 1969, pp. 239-307.
68. James C. Beatty, "An Axiomatic Approach to Code Optimization for Expressions," *J. ACM* **19**, 613-640 (1972). Errata: *J. ACM* **20**, 188 (1973) and *J. ACM* **20**, 538 (1973).
69. J. C. Beatty, "A Global Register Assignment Algorithm," *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972, pp. 65-88.
70. Frances E. Allen, "Control Flow Analysis," *Proceedings ACM SIGPLAN Symposium on Compiler Optimization, SIGPLAN Notices* **5**, 1-19 (July 1970).
71. J. Cocke, "Global Common Subexpression Elimination," *Proceedings ACM SIGPLAN Symposium on Compiler Optimization, SIGPLAN Notices* **5**, 20-24 (July 1970).
72. Robert Tarjan, "Testing Flow Graph Reducibility," *J. Computer Syst. Sci.* **9**, 355-365 (1974).
73. F. E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972, pp. 1-30.
74. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM* **19**, 137-147 (1976).
75. Susan L. Graham and Mark Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," *J. ACM* **23**, 172-202 (1976).
76. F. E. Allen, "Interprocedural Data Flow Analysis," *Proc. IFIP Congress 74*, North-Holland Publishing Co., Amsterdam, 1974, pp. 398-402.
77. T. C. Spillman, "Exposing Side Effects in a PL/I Optimizing Compiler," *Proc. IFIP Congress 71*, North-Holland Publishing Co., Amsterdam, 1971, pp. 376-381.
78. B. K. Rosen, "Data Flow Analysis for Procedural Languages," *J. ACM* **26**, 322-344 (1979).
79. Jeffrey M. Barth, "Interprocedural Data Flow Analysis Based on Transitive Closure," *Conference Record of the Fourth ACM Symposium On Principles of Programming Languages*, Santa Monica, CA, January 1977, pp. 119-131.
80. John Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and Aliases of Variables," *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979, pp. 29-41.
81. F. E. Allen, J. L. Carter, J. Fabri, J. Ferrante, W. H.

Harrison, P. G. Loewner, and L. H. Trevillyan, "The Experimental Compiling System," *IBM J. Res. Develop.* **24,** 695-715 (1980).

82. William Harrison, "A New Strategy for Code Generation—the General Purpose Optimizing Compiler," *IEEE Trans. Software Engineering* **SE-3,** 243-250 (1977).
83. William E. Weihl, "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables," *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 83-94.
84. D. B. Lomet, "Data Flow Analysis in the Presence of Procedure Calls," *IBM J. Res. Develop.* **21,** 559-571 (1977).
85. W. H. E. Day, "Compiler Assignment of Data Items to Registers," *IBM Syst. J.* **9,** 281-317 (1970).
86. R. Sethi, "Complete Register Allocation Problems," *SIAM J. Computing* **4,** 226-248 (1975).
87. John Cocke and Peter W. Markstein, "Measurement of Program Improvement Algorithms," *IFIP 80 Proceedings Information Processing*, S. H. Lavington, Ed., North-Holland Publishing Co., Amsterdam, 1980, pp. 221-228.
88. Jacob T. Schwartz, "On Programming (An Interim Report on the SETL Project) Installment I: Generalities," *Technical Report*, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, New York, 1973.
89. Donald E. Knuth, "A History of Writing Compilers," *Computers and Automation* **11,** 8-14 (1962).

Received October 23, 1980; revised February 24, 1981

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.