
# THE FORTRAN I COMPILER

*The Fortran I compiler was the first demonstration that it is possible to automatically generate efficient machine code from high-level languages. It has thus been enormously influential. This article presents a brief description of the techniques used in the Fortran I compiler for the parsing of expressions, loop optimization, and register allocation.*

During initial conversations about the topic of this article, it became evident that we can't identify *the* top compiler algorithm of the century if, as the *CiSE* editors originally intended, we consider only the parsing, analysis, and code optimization algorithms found in undergraduate compiler textbooks and the research literature. Making such a selection seemed, at first, the natural thing to do, because fundamental compiler algorithms belong to the same class as the other algorithms discussed in this special issue. In fact, fundamental compiler algorithms, like the other algorithms in this issue, are often amenable to formal descriptions and, as a result, to mathematical treatment.

However, in the case of compilers the difficulty is that, paraphrasing John Donne, no algorithm is an island, entire of itself. A compiler's components are designed to work together to complement each other. Furthermore, next to this conceptual objection, there is the very practical issue that we don't have enough information to decide whether any of the fundamental compiler algorithms have had a determinant impact on the quality of compilers.

1521-9615/00/$10.00 © 2000 IEEE

DAVID PADUA
*University of Illinois at Urbana-Champaign*

At the same time, it is almost universally agreed that the most important event of the 20th century in compiling—and in computing—was the development of the first Fortran compiler between 1954 and 1957. By demonstrating that it is possible to automatically generate quality machine code from high-level descriptions, the IBM team led by John Backus opened the door to the Information Age.

The impressive advances in scientific computing, and in computing in general, during the past half century would not have been possible without high-level languages. Although the word *algorithm* is not usually used in that sense, from the definition it follows that a compiler *is* an algorithm and, therefore, we can safely say that the Fortran I translator is the 20th century's top compiler algorithm.

## The language

The IBM team not only developed the compiler but also designed the Fortran language, and today, almost 50 years later, Fortran is still the language of choice for scientific programming. The language has evolved, but there is a clear family resemblance between Fortran I and today's Fortran 77, 90, and 95. Fortran's influence is also evident in the most popular languages today, including numerically oriented languages

such as Matlab as well as general-purpose languages such as C and Java.

Ironically, Fortran has been the target of criticism almost from the beginning, and even Backus voiced serious objections: "'von Neuman languages' [like Fortran] create enormous, unnecessary intellectual roadblocks in thinking about programs and in creating the higher-level combining forms required in a powerful programming methodology."[1]

Clearly, some language features, such as implicit typing, were not the best possible choices, but Fortran's simple, direct design enabled the development of very effective compilers. Fortran I was the first of a long line of very good Fortran compilers that IBM and other companies developed. These powerful compilers are perhaps the single most important reason for Fortran's success.

## The compiler

The Fortran I compiler was fairly small by today's standards. It consisted of 23,500 assembly language instructions and required 18 person-years to develop. Modern commercial compilers might contain 100 times more instructions and require many more person-years to develop. However, its size not withstanding, the compiler was a very sophisticated and complex program. It performed many important optimizations—some quite elaborate even by today's standards—and it "produced code of such efficiency that its output would startle the programmers who studied it."[1]

However, as expected, the success was not universal.[2] The compiler seemingly generated very good code for regular computations; however, irregular computations, including sparse and symbolic computations, are generally more difficult to analyze and transform. Based on my understanding of the techniques used in the Fortran I compiler, I believe that it did not do as well on these types of computations. A manifestation of the difficulties with irregular computations is that subscripted subscripts, such as `A(M(I,J),N(I,J))`, were not allowed in Fortran I.

The compiler's sophistication was driven by the need to produce efficient object code. The project would not have succeeded otherwise. According to Backus:

> It was our belief that if Fortran, during its first months, were to translate any reasonable scientific source program into an object program only half as fast as its hand-coded counterpart, the acceptance of our system would be in serious danger.[1]

The flip side of using novel and sophisticated compiler algorithms was implementation and debugging complexity. Late delivery and many bugs created more than a few Fortran skeptics, but Fortran eventually prevailed:

> It gradually got to the point where a program in Fortran had a reasonable expectancy of compiling all the way through and maybe even of running. This gradual change in status from an experimental to a working system was true of most compilers. It is stressed here in the case of Fortran only because Fortran is now almost taken for granted, as if it were built into the computer hardware.[2]

## Optimization techniques

The Fortran I compiler was the first major project in code optimization. It tackled problems of crucial importance whose general solution was an important research focus in compiler technology for several decades. Many classical techniques for compiler analysis and optimization can trace their origins and inspiration to the Fortran I compiler. In addition, some of the terminology the Fortran I implementers used almost 50 years ago is still in use today. Two of the terms today's compiler writers share with the 1950s IBM team are *basic block* ("a stretch of program which has a single entry point and a single exit point"[3]) and *symbolic/real registers*. Symbolic registers are variable names the compiler uses in an intermediate form of the code to be generated. The compiler eventually replaces symbolic registers with real ones that represent the target machine's registers.

Although more general and perhaps more powerful methods have long since replaced those used in the Fortran I compiler, it is important to discuss Fortran I methods to show their ingenuity and to contrast them with today's techniques.

### Parsing expressions

One of the difficulties designers faced was how to compile arithmetic expressions taking into account the precedence of operators. That is, in the absence of parentheses, exponentiation should be evaluated first, then products and divisions, followed by additions and subtractions. Operator precedence was needed to avoid extensive use of parentheses and the problems associated with them. For example, IT, an experimental compiler completed by A. Perlis and J.W. Smith in 1956 at the Carnegie Institute of Technology,[4] did not assume operator precedence. As

Donald Knuth pointed out: "The lack of operator priority (often called precedence or hierarchy) in the IT language was the most frequent single cause of errors by the users of that compiler."[5]

The Fortran I compiler would expand each operator with a sequence of parentheses. In a simplified form of the algorithm, it would

- replace + and – with ))+(( and ))-((, respectively;
- replace * and / with )*( and )/(, respectively;
- add (( at the beginning of each expression and after each left parenthesis in the original expression; and
- add )) at the end of the expression and before each right parenthesis in the original expression.

***It is interesting to contrast the parsing algorithm of Fortran I with more advanced parsing algorithms.***

Although not obvious, the algorithm was correct, and, in the words of Knuth, "The resulting formula is properly parenthesized, believe it or not."[5] For example, the expression A + B * C was expanded as ((A))+((B)*(C)). The translation algorithm then scanned the resulting expression from left to right and inserted a temporary variable for each left parenthesis. Thus, it translated the previous expression as follows:

$u_1=u_2+u_4;\ \ u_2=u_3;\ \ u_3=A;\ \ u_4=u_5*u_6;$
$u_5=B;\ u_6=C.$

Here, variable $u_i$ $(2 \le i \le 5)$ is generated when the $(i-1)$th left parenthesis is processed. Variable $u_1$ is generated at the beginning to contain the expression's value. These assignment statements, when executed from right to left, will evaluate the original expression according to the operator precedence semantics. A subsequent optimization eliminates redundant temporaries. This optimization reduces the code to only two instructions:

$u_1=A+u_4;\ u_4=B*C.$

Here, variables A, B, and C are propagated to where they are needed, eliminating the instructions rendered useless by this propagation.

In today's terminology, this optimization was equivalent to applying, at the expression level, *copy propagation* followed by *dead-code elimination*.[6] Given an assignment x = a, copy propagation substitutes a for occurrences of x whenever it can determine it is safe to do so. Dead-code elimination deletes statements that do not affect the program's output. Notice that if a is propagated to all uses of x, x = a can be deleted.

The Fortran I compiler also identified permutations of operations, which reduced memory access and eliminated redundant computations resulting from common subexpressions.[3] It is interesting to contrast the parsing algorithm of Fortran I with more advanced parsing algorithms developed later on. These algorithms, which are much easier to understand, are based on syntactic representation of expressions such as:[7]

*expression* = *term* [ [ + | –] *term*]...
*term* = *factor* [ [ * | / ] *factor*]...
*factor* = *constant* | *variable* | (*expression* )

Here, a *factor* is a *constant*, *variable*, or *expression* enclosed by parentheses. A *term* is a factor possibly followed by a sequence of factors separated by * or /, and an expression is a term possibly followed by a sequence of terms separated by + or –. The precedence of operators is implicit in the notation: terms (sequences of products and divisions) must be formed before expressions (sequences of additions and subtractions). When represented in this manner, it is easy to build a recursive descent parser with a routine associated with each type of object, such as term or factor. For example, the routine associated with term will be something like

```
procedure term(){
       call factor()
       while token is * or / {
               get next token
               call factor()
               }
       }
```

Multiplication and division instructions could be generated inside the while loop (and addition or subtraction in a similar routine written to represent expressions) without redundancy, thus avoiding the need for copy-propagation or dead-code-elimination optimization within an expression.

### DO loop optimizations and subscript computations

One of the Fortran I compiler's main objectives was "to analyze the entire structure of the program in order to generate optimal code from DO state-

ments and references to subscripted variables."[1] For example, the address of the Fortran array element `A(I,J,c3*K+6)` could take the form

$$\texttt{base\_A+I-1+(J-1)*}d_i\texttt{+(}c_3\texttt{*K+6-1)*}d_i\texttt{*}d_j$$

where $d_i$ and $d_j$ are the length of the first two dimensions of `A`, and these two values as well as the coefficient $c_3$ are assumed to be constant. Clearly, address expressions such as this can slow down a program if not computed efficiently.

It is easy to see that there are constant subexpressions in the address expression that can be incorporated in the address of the instruction that makes the reference.[3] Thus, an instruction making reference to the previous array element could incorporate the constant $\texttt{base\_A+(6-1)*}d_i\texttt{*}d_j\texttt{-}d_i\texttt{-1}$. It is also important to evaluate the variant part of the expression as efficiently as possible. The Fortran I compiler used a pattern-based approach to achieve this goal. For the previous expression, every time "`K` is increased by `n` (under control of a `DO`), the index quantity is increased by $c_3 d_i d_j n$, giving the correct new value."[3]

Today's compilers apply *removal of loop invariants*, *induction-variable detection*, and *strength reduction* to accomplish similar results.[6,8] The idea of induction-variable detection is to identify those variables within a loop that assume a sequence of values forming an arithmetic sequence. After identifying these induction variables, strength reduction replaces multiplications of induction-variable and loop-invariant values with additions.

The Fortran I compiler applied, instead, a single transformation that simultaneously moved subexpressions to the outermost possible level and applied strength reduction. A limitation of the Fortran I compiler, with respect to modern methods, was that it only recognized loop indices as induction variables:

> It was decided that it was not practical to track down and identify linear changes in subscripts resulting from assignment statements. Thus, the sole criterion for linear changes, and hence for efficient handling of array references, was to be that the subscripts involved were being controlled by `DO` statements.[1]

### Register allocation

The IBM 704, the Fortran I compiler's target machine, had three index registers. The compiler applied register allocation strategies to reduce the number of load instructions needed to bring values to the index registers. The compiler section that Sheldon Best designed, which performed index-register allocation, was extremely complex and probably had the greatest influence on later compilers.[1] Indeed, seven years after Fortran I was delivered, Saul Rosen wrote:

> Part of the index register optimization fell into disuse quite early, but much of it was carried along into Fortran II and is still in use on the 704/9/90. In many programs it still contributes to the production of better code than can be achieved on the new Fortran IV compiler.[2]

The register allocation section was preceded by another section whose objective was to create what today is called a control-flow graph. The nodes of this graph are basic blocks and its arcs represent the flow of execution. Absolute execution frequencies were computed for each basic block using a Monte Carlo method and the information provided by Frequency statements. Fortran I programmers had to insert the Frequency statements in the source code to specify the branching probability of `IF` statements, computed `GOTO` statements, and average iteration counts for `DO` statements that had variable limits.[9]

***Although the Monte Carlo algorithm delivered the necessary results, not everybody liked the strategy.***

Compilers have used Frequency information for register allocation and for other purposes. However, modern compilers do not rely on programmers to insert information about frequency in the source code. Modern register allocation algorithms usually estimate execution frequency using syntactic information such as the level of nesting. When compilers used actual branching frequencies, as was the case with the Multiflow compiler,[10] they obtained the information from actual executions of the source program.

Although the Monte Carlo Algorithm delivered the necessary results, not everybody liked the strategy:

> The possibility of solving the simultaneous equations determining path frequency in terms of transition frequency using known methods for solving sparse matrix equations was considered, but no methods which would work in the presence of `DO`-loops and assigned `GOTO` statements [were] hit upon, although `IF`-type branches alone could

> have been handled without explicit interpretation. The frequency estimating simulation traced the flow of control in the program through a fixed number of steps, and was repeated several times in an effort to secure reliable frequency statistics. Altogether an odd method![9]

With the estimated value of execution frequency at hand, the compiler proceeded to create *connected regions*, similar to the traces used many years later in the Multiflow compiler. Regions were created iteratively. In each iteration, the control flow graph was scanned one at a time by finding at each step the basic block with the highest absolute execution frequency. Then, working backwards and forward, a chain was formed by following the branches with the highest probability of execution as specified in the Frequency statements. Then, registers were allocated in the new region

> … by simulating the action of the program. Three cells are set aside to represent the object machine index registers. As each new tagged instruction is encountered, these cells are examined to see if one of them contains the required tag; if not, the program is searched ahead to determine which of the index registers is the least undesirable to replace.[3]

The new regions could connect with old regions and subsume them into larger regions.

> In processing a new path connecting two previously disconnected regions, register usage was matched by permuting all the register designations of one region to match those of the other as necessary.[9]

The process of dealing with loops was somewhat involved.

> In processing a new path linking a block to itself and thus defining a loop, the loop was first considered to be concatenated with a second copy of itself, and straight-line register allocation carried out in normal fashion through the first of the two copies, with look-ahead extending into the second copy. …Straight-line allocation was carried out for a second loop copy in essentially normal fashion.[9]

The only difference was that the look-ahead procedure employed during this allocation was a modified version of the original look-ahead procedure to account for register reuse across loop iterations. Finally, "the allocation produced for the second loop was that ultimately used in generating machine code."[9]

The "least undesirable" register the look-ahead procedure identified was one whose value was dead or, if all registers were live, the one reused most remotely within the region. This strategy is the same as that proved optimal by Laszlo A. Belady in a 1965 paper for page replacement strategies.[11] Belady's objective was to minimize the number of page faults; as a result, the algorithm is optimal "as long as one is concerned only with minimizing the number of loads of symbolic indexes into actual registers and not with minimizing the stores of modified indexes."[9]

The goal, of course, was not to prove or even achieve optimality of the register allocation algorithm. In fact,

> [i]n order to simplify the index register allocation, it was implicitly assumed that calculations were not to be reordered. The contrary assumption would have introduced a new order of difficulty into the allocation process, and required the abstraction of additional information from the program to be processed.[9]

This assumption meant that the result is not always optimal because, in some cases, "… there is much advantage to be had by reordering computations."[7] Nevertheless, "… empirically, Best's 1955–1956 procedure appeared to be optimal."[1]

During the last decade, the relative importance of traditional programming languages as the means to interact with computers has rapidly declined. The availability of powerful interactive applications has made it possible for many people to use computers without needing to write a single line of code.

Although traditional programming languages and their compilers are still necessary to implement these applications, this is bound to change. I do not believe that 100 years hence computers will still be programmed the same way they are today. New applications-development technology will supersede our current strategies that are based on conventional languages. Then, the era of compilers that Fortran I initiated will come to an end.

Technological achievements are usually of interest for a limited time only. New techniques or devices rapidly replace old ones in an endless

cycle of progress. All the techniques used in the Fortran I compiler have been replaced by more general and effective methods. However, Fortran I remains an extraordinary achievement that will forever continue to impress and inspire.

## Acknowledgments

*This work was supported in part by US Army contract N66001-97-C-8532; NSF contract ACI98-70687; and Army contract DABT63-98-1-0004. This work is not necessarily representative of the positions or policies of the Army or Government.*

## References

1. J. Backus, "The History of Fortran I, II, and III," *IEEE Annals of the History of Computing*, Vol. 20, No. 4, 1998.
2. S. Rosen, "Programming Systems and Languages—A Historical Survey," *Proc. Eastern Joint Computer Conf.*, Vol. 25, 1964, pp. 1–15, 1964; reprinted in *Programming Systems and Languages*, S. Rosen, ed., McGraw-Hill, New York, 1967, pp. 3–22.
3. J.W. Backus et al., "The Fortran Automatic Coding System," *Proc. Western Joint Computer Conf.*, Vol. 11, 1957, pp. 188–198; reprinted in *Programming Systems and Languages*, S. Rosen, ed., McGraw-Hill, New York, 1967, pp. 29–47.
4. D.E. Knuth and L.T. Pardo, "Early Development of Programming Languages," *Encyclopedia of Computer Science and Technology*, Vol. 7, Marcel Dekker, New York, 1977, p. 419.
5. D.E. Knuth, "A History of Writing Compilers," 1962; reprinted in *Compiler Techniques*, B.W. Pollack, ed., Auerbach Publishers, Princeton, N.J., 1972, pp. 38–59.
6. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1988.
7. W.M. McKeeman, "Compiler Construction," *Compiler Construction: An Advanced Course*, F.L. Bauer and J.Eickel, eds., Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, Berlin, 1976.
8. S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.
9. J. Cocke and J.T. Schwartz, *Programming Languages and Their Compilers, Courant Inst. of Mathematical Sciences*, New York Univ., New York, 1970.
10. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, Vol. C-30, No. 7, July 1981, pp. 478–490.
11 L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems J.*, Vol. 5, No. 2, 1966, pp. 78–101.

**David Padua** is a professor of computer science at the University of Illinois, Urbana-Champaign. His interests are in compiler technology, especially for parallel computers, and machine organization. He is a member of the ACM and a fellow of the IEEE. Contact him at the Dept. of Computer Science, 3318 Digital Computer Laboratory, 1304 W. Springfield Ave., Urbana, IL 61801; padua@uiuc.edu; polaris.cs.uiuc.edu/~padua.

## *Classified Advertising*

**PURDUE UNIVERSITY**
**Director**
**Computing Research Institute**

Purdue University seeks a person with vision to direct its Computing Research Institute. The Institute's mission is to enhance computing research across the Purdue campus by providing focus for interdisciplinary initiatives and by facilitating large innovative research programs. The University is prepared to invest resources to achieve this end.

**Position Responsibilities.** The Direct-or will provide leadership for activities designed to promote and facilitate research in computer systems and their applications in science and engineering; lead development of collaborative relationships with government and industry; assist faculty in identifying research opportunities; and organize multidisciplinary research programs.

The appointment is full-time with a minimum half-time effort devoted to Institute leadership. Remaining effort may include responsibilities in research, education, and administrative duties.

**Requirements.** Applicants must have a Ph.D. in a science or engineering discipline and must have demonstrated effective leadership of multi-investigator research programs. Administrative and academic experience should be commensurate with an appointment to a Full Professor faculty position.

The Director will report to the Vice President for Research and Dean of the Graduate School.

**Applications/Nominations.** A review of applications will commence December 1, 1999, and continue until the position is filled. Nominations or applications containing a résumé and names of three references should be sent to:

Dr. Richard J. Schwartz,
Chair of the Search Committee
Dean, Schools of Engineering
Purdue University
1280 ENAD Building
West Lafayette, In 47907-1280
PHONE: 764-494-5346;
FAX: 765-494-9321

Purdue University is an Equal Opportunity/Affirmative Action Employer.

---

**CUNY GRADUATE CENTER**

Computer Science: Ph.D. Program in Computer Science at CUNY Graduate Center has two professorial positions, possibly at Distinguished Professor level. Seek individuals who have had major impact in computer science, are active in more than one area, have consistent grant record, and some of whose work is applied. Candidates with wide-ranging computational science backgrounds (computer science, computational biology, chemistry, physics, mathematics, or engineering) as well as interests in new media are encouraged to apply. Candidates will have opportunity to be associated with CUNY Institute for Software Design and Development and New Media Lab at Graduate Center. See http://www.cuny.edu/abt-cuny/cunyjobs. Send CV and names/addresses of three references by 1/31/00 to: Search Committee Chair, Ph.D. Program in Computer Science, CUNY Graduate Center, 365 Fifth Avenue, New York, NY 10016. EO/AA/IRCA/ADA