Programming in *Standard* FORTRAN by Functional Blocks: A Rigorous *Structured* Approach

Ronald C. Wackwitz and MayNell H. Wackwitz
The COMP-AID Company
513 Old Bear Creek Road
New Braunfels, TX 78132

Original version: May, 1977 A.D.
Revised May, 2006 A.D.

Contents

List of Tables

1 Introduction

1.1 Superior Structured Coding with *Standard* FORTRAN

We at COMP-AID have been using *standard* FORTRAN to achieve *rigorously* structured code for 28 years now. Instead of denoting structure through use of syntactic constructs (which standard FORTRAN[1] lacks), we denote structure through our method of *Functional Blocks*.

We advance in this paper the argument that equal, *if not superior*, overall programming efficiency results from the use of Functional Blocks, rather than from the use of a"structured" language (e.g., the Fortran 90 series, PL/I, ALGOL, PASCAL, etc.), simply because the use of Functional Blocks requires the interactive involvement of the programmer — a criterion Prof. Donald Knuth rates as highly desirable.[34][2]

1.2 A Rigorous Approach

We present evidence to show that the use of Functional Blocks in standard FORTRAN results in code *which is verifiable* - i.e., which is amenable to proof-of-correctness. Modules[3] coded in this fashion are highly understandable, and as a result are easily maintained. Moreover, since the coding it performed in standard FORTRAN, the execution speed of the resulting code is superior to that of a comparable module coded in a high-level structured language.

One of the main purposes of this paper is to prove that the use of Functional Blocks in coding is a rigorous approach. Accordingly, this paper presupposes that the reader already is familiar with our concept of Functional Blocks. To the reader who is not, we recommend our companion report, *Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)*[41],[4] where the concept is carefully explained and vividly illustrated. However, in the Appendix to this paper, we also present a Very brief overview of our concept of Functional Blocks.

[1] By this, we mean all the FORTRANs up to FORTRAN 77, but excluding the Fortran 90 series.

[2] The authors disclaim any endorsement of this position by either Profs. Knuth or Dijkstra. However, we feel we have in every instance quoted Prof. Knuth *completely in context*. Moreover, every quote is carefully referenced to its very page of use.

[3] The term "module", up through Fortran 77, denoted a unit of source code (e.g., a program, a subroutine, a function, or a block data unit of source code). Starting with Fortran 90, however, the term was preempted to denote a structure-type container, completely disregarding years of prior usage up through FORTRAN 77. In this paper, we use the term *module* in the original sense, unless otherwise noted.

[4] Until we are able to post this report onto the Internet, please refer to our related report, *Overview of the Methodology of Functional Blocks*, which is posted on the Internet.

1.3 A Dual Reaction

We envision a dual reaction to our paper. Of course, that which we stated back in 1977, when this report was first published, will differ somewhat from that which is applicable for 2006. We shall give both:

- **Back in 1977.**
  1. We expect that the zealous advocates of the New Programming will find our paper to be somewhat disquieting at the least, if not outright shocking at the most, since it has never occurred to many of them that there may be equally good alternatives to their vision of structuring code by syntactic constructs. Nor will it soothe them any to learn that our methodology is highly consistent with the views espoused by Prof. Donald Knuth, and even with some of the later views of Prof. E.W. Dijkstra.
  2. From the vast majority of users in Industry who still use standard FORTRAN, however, we expect a much happier reaction. Many of these users are already understandably upset that they have been branded sinners by the advocates of the new Programming for not abandoning standard FORTRAN. Now they can brush aside these indictments and continue to get their work done, leaving the academic philosophizing to their accusors.
- **Fast forward to 2006.**
  1. By and large, the so-called structured code of the 1970's and 1980's has been replaced by the Object Orientated Programming Languages (**OOPLs**). And even FORTRAN itself has been preempted with the appearance of the Fortran 90 series of languages. Regarding Fortran 90, Andrew Scriven[43] states:

     > Fortran 90 is truly so different from previous Fortran that adopting it as the language of choice is really like changing languages. ... The proliferation of new features recalls the old joke: "The programming language of the future will be nothing like Fortran and will be called Fortran!"

     The abundance of syntactic constructs in the Fortran 90 series permits one to program *without* using any statement numbers. Moreover, the fact that the Fortran X3J3 committee within Fortran 90 is suggesting obsoleting the `GOTO` statement once again strongly suggests that the committee views the "old FORTRANs" (FORTRAN 77 and downward) as inferior and obsolete languages.
  2. We readily admit that the level of abstraction is greater in Fortran 90 where vector and matrix operations are involved. However, since the use of Functional Blocks within FORTRAN 77 raises the *overall* level of abstraction, we suggest that the use of FORTRAN 77 with Functional Blocks, when reclaiming legacy FORTRAN code, is much more efficient than first processing the legacy FORTRAN code through one of the many `GOTO` removers to convert it to Fortran 90. No, first structure (reclaim or refactor) the module by Functional Blocks, next modify (upgrade) the code as desired, and *then* convert it to the desired final language (e.g., Fortran 90 or C++). Moreover, this approach is also consistent with Prof. Knuth's warning (cf. Section 4.8 on page 13) that the removal of `GOTO`s from a poorly structured original module will only result in a poorly structured transformed module.

2 The software impasse: it's impact, cause, and suggested cure

2.1 A Software Impasse

We have reached an impasse in our software development efforts. *overrun budgets*, *missed dead lines*, and *unmet specifications* are the rule in most DP departments, rather than the exception.

To appreciate the magnitude of this indictment, we turn to an Air Force study, CCIP-85, which Dr. Barry W. Boehm made public in 1973.[1] We find that the annual Air Force software budget during the previous year (in 1972) was between $1 billion and $1.5 billion, making it about three times as great as the Air Force expenditure on computer hardware for that year, and about 4 to 5% of the *total* Air Force budget! Study CCIP-85 estimated software expenses in the Air Force rising to *over 90%* of its total DP budget by 1985. Dr. Boehm considers that this trend is probably characteristic of other organizations also.

From a different perspective, we learn that the "average design and programming cost per instruction" rose from $4.50 in 1959 to over $7.50 in 1975.[2] Specific costs have been much higher. For example, the programs used in the Apollo moon missions cost as much as $200 per instruction.[3]

Moreover, according to Charles P. Lecht in his upcoming book, *The waves of Change*, "if current trends continue, by 1980 it is estimated that more than 80% of the computer user's total human resources — programmers and system analysts — will be devoted to the maintenance of old application programs, with less than 20% of these resources available for new applications development."[4]

2.2 Suggested Causes

What has been responsible for this software crisis? *Lack of communication* with the *end user* would undoubtedly be high on the list of suggested causes. To this we must also add at least three additional causes: (1) inadequate program design, (2) unverifiable code, and (3) lack of effective program management.

2.3 Suggested Solutions

For each cause listed above, successfully-tried methodologies either have evolved, or are in the process of evolving, to correct the associated abuses, as shown in Table 1 on the following page.

Table 1: Suggested Solutions for the Software Impasse

Cause	**Suggested Solutions**
Lack of communication with end user	HIPO charts[5][6] Structured Walkthroughs[6] (or Design Reviews[7])
Inadequate program design	Decomposition of program into small independent functional modules[8][9] Chief Programmer Team[10][11] Top-Down Development[10][11] Stepwise Refinement[44]
Unverifiable code	Structured Code[12][13] "Egoless" Programmers[14][15][16] Decomposition of program into small independent functional modules[8][9] Code Reviews[6]
Lack of effective program management	Chief Programmer Team[10][11] Top-Down Development[10][11] Stepwise Refinement[44]

2.4 Language Independence

Of the suggested solutions presented above, *all are independent of the programming language*, except for one[5]: *Structured Code.*

In the case of structured code, however, language has been deemed *strongly* important. In fact, the concensus of some authorities has been that certain languages, which they feel do not properly support the "structured code" concept, should accordingly be *disregarded*! Standard FORTRAN is one such of these.

2.5 A Prerequisite to Structured Coding

In the remainder of this paper we shall concern ourselves exclusively with structured code - its "definition", background, and application to standard FORTRAN. However, we must realize that structured code, as just *one* of the new methodologies, is not an end-in-all in itself. It depends, as a prerequisite, both upon *program design* and *user acceptance* having been satisfactorily completed — at least *initially*, keeping in mind Wirth's[44] reminder that stepwise refinement will allow the user's requirements to further be *clarified* as coding, testing, and user review progress. Even Prof. Dijkstra reflected on this in his reply to Van der Poel[18] that "completely specified problems" have a way of subsequently being altered by the user. So we see that the requirements of Wirth and Dijkstra, back in 1977, are very similar to that of the "Extreme Programming"[6] of 2006. Isn't it interesting, as typified by the old saying, "What goes around, comes around", that the "old" methodologies of the 1970's reappear today, albeit with just "new" names.

[5] G.J. Myers has recently[17] pointed out that the ability to decompose a program into small independent functional modules is also dependent upon language, with PL/I and FORTRAN being the most amenable to this concept.

[6] Please see "How Agile Are You?"[45] for an excellent brief review of the Extreme Programming methodology — this one from the standpoint of programming in OOPLs. And, for an even better review — not only because it is pertinent to Fortran-95 rather than to JAVA or C++, but also because it most adequately exemplifies the methodology — please see the NASA Langley Research Center publication by William Kleb *et al*[46].

2.6 Structured Coding / Structured Programing

The terms *Structured Coding* and *Structured Programming* are at times used interchangeably to denote the same process. At other times *Structured Programing* is used to denote *all* the new methodologies, while *Structured Coding* simply refers to a method for achieving understandable and verifiable code within given modules. In this paper we shall accordingly concentrate on that aspect of Structured Programming which deals with coding individual modules in an understandable and verifiable fashion.

3 Structured Programming: It's Meaning, and it's Goals

3.1 Structured Programming Equated to GOTO-less Programming

If we're truthful, most of us must admit that we equate structured programing to GOTO-less programming.

From a historical perspective,[7] doubt concerning the use of the GOTO began building as early as 1959, culminating in 1968 with Prof. Dijkstra's famous paper, "**Go to** statement considered harmful". Concerning this paper, Prof. Donald Knuth comments[20]:

> This note[12] rapidly became well-known; it expressed Dijkstra's conviction that **go to's** 'should be abolished from all higher level programming languages (i.e., everything except, perhaps, plain machine code) .... The **go to** statement as is stands is just too primitive; it is too much an invitation to make a mess of one's program.' He encouraged looking for alternate constructions which may be necessary to satisfy all needs.

In that one paper we have the two main objections to the use of GOTO's:

1. *Undisciplined* use of them can lead to unintelligible spaghetti-bowl code
2. Code utilizing GOTO's was considered to not be amenable to proof-of-correctness.

3.2 Real Issue Is Not Removal of GOTO's

While many would agree that the real issue is *structured programming*, yet, as Prof. Knuth notes,[21]

> ... unfortunately this has become a catch phrase whose meaning is rarely understood in the same way by different people.... Only one thing is really clear: Structured programming is *not* the process of writing programs and then eliminating their **go to** statements.... Indeed, Dijkstra's original article[22] which gave Structured Programming its name never mentions **go to** statements at all; he directed attention to the critical question, 'For what program structures can we give correctness proofs without undue labor, even if the programs get large?' By correctness proofs he explained that he does not mean formal derivations from axioms, he means any sort of proof (formal or informal) that is 'sufficiently convincing'; and a proof really means an understanding.

To those of us who have equated structured programming to GOTO-less programming, Prof. Knuth's above statement may be somewhat disquieting. Continuing in this vain, that the real issue is not the removal of GOTO'S, Knuth states[23]:

[7] An excellent historical background of the events leading to the current disfavor with GOTO statements has been presented by Prof. Knuth.[19]

> In other words, we shouldn't merely remove **go to** statements because it's the fashionable thing to do; the presence or absence of **go to** statements is not really the issue. The underlying structure of the program is what counts, and we want only to avoid usages which somehow clutter up the program. Good structure can be expressed in FORTRAN or COBOL, or even in assembly language, although less clearly and with much more trouble. The real goal is to formulate our programs in such a way that they are easily understood.

Premise 1: The real goal is to formulate our programs in such a way that they are easily understood.

3.3 GOTO's Pemissable as Verifiable Constructs

Prof. Knuth has just delivered another disquieting pronouncement - that good structure can be expressed in FORTRAN or COBOL, or even in assembly language, albeit "less clearly and with much more trouble".

Interestingly, we've since realized that the GOTO construct, used under certain well-defined conditions, is amenable to proof-of-correctness, as Prof. Knuth notes[24]:

> For many years, the **go to** statement has been troublesome in the definition of correctness proofs and language semantics; .... Just recently, however, Hoare has shown that there is, in fact, a rather simple way to give an axiomatic definition of **go to** statements; indeed, he wishes quite frankly that it hadn't been quite so simple. For each label `L` in a program, the programmer should state a logical assertion a(L) which is to be true whenever we reach `L`. Then the axioms
>
> {a(L)} **go to** L {false}
>
> plus the rules of inference
>
> {a(L)} S{P} $\vdash$ {a(L)} `L`:S{P}
>
> are allowed in program proofs, and all properties of labels and **go to**'s will follow if the a(L) are selected intelligently. One must, of course, carry out the entire proof using the same assertion a(L) for each appearance of the label `L`, and some choices of assertions will lead to more powerful results than others.
>
> Informally, a(L) represents the desired state of affairs at label `L`; this definition says essentially that a program is correct if a(L) holds at `L` and before all '**go to** `L`' statements, and that control never 'falls through' a **go to** statement to the following text. Stating the assertions a(L) is analogous to formulating loop invariants. Thus, it is not difficult to deal formally with tortuous program structure if it turns out to be necessary; all we need to know is the 'meaning' of each label.

Premise 2: The GOTO construct is allowed in program proofs if, for each label `L`, there is a logical assertion a(L) which is to be uniquely true, *for each appearance of the label* `L`, whenever we reach `L`.

Corollary: Programs written in *Standard* FORTRAN, utilizing the method

of Functional Blocks, are verifiable.

Corollary: Maintenance changes made to a Functional Block (in which only "proper GOTO branching" is utilized) will be *isolated* to that block.

3.4 Many Good Programs over the Years Had GOTO's

Knuth, continuing, states:[25]:

> In other words, we can indeed consider **go to** statements as part of systematic abstraction; all we need is a clearcut notion of exactly what it means to **go to** each label. This should come as no great surprise. After all, a lot of computer programs have been written using **go to** statements during the last 25 years, and these programs haven't all been failures! Some programmers have clearly been able to master structure and exploit it; not as consistently, perhaps, as in modern-day structured programming, but not inflexibly either. By now, many people who have never had any special difficulty writing correct programs have naturally been somewhat upset after being branded as sinners, especially when they know perfectly well what they're doing; so they have understandably been less than enthusiastic about 'structured programing' as it has been advertised to them.

Premise 3: A powerful alternative to structuring code by syntactic constructs does exist: structuring code by Functional Blocks.

Corollary: Programs exhibiting a high level of understandability can be programmed in *Standard* FORTRAN utilizing the method of Functional Blocks.

4 The Skeletons-in-the-Closet of the New Programming

4.1 Methodology Alone Is Insufficient

Prof. Knuth's comments which we have presented above concerning use of the GOTO do not disagree sharply with Prof. Dijkstra's more recent views. Concerning this, Knuth states[26]:

> I believe that by presenting such a view I am not in fact disagreeing sharply with Dijkstra's ideas, since he recently wrote the following: 'Please don't fall into the trap of believing that I am terribly dogmatical about [the **go to** statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline![27]

We should like to emphasize Prof. Dijkstra's warning: "... as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!" Others have offered the same pronouncements. For example, Richard J. Weiland[28] bears witness that "... the unfortunate fact is that programs written in PL/I ... are typically as crummy and unstructured as those in any other language.... Obviously, this is not to say that one can't write handsomely structured PL/I, just that not very many people do. Alas, *Simply picking PL/I coding is not enough.*[8]" Weiland concludes that the two main requirements, (1) "thinking through the transformations that produce required results," and (2) "developing a functional design", are really independent of whether "one is going to code in COBOL, PL/I, FORTRAN, Assembler, or LISP".

The witness of Kenneth T. Orr[29] is similar. He feels that "if one attacks a problem logically and with the right design tools, the language used to code the problem becomes considerably less important." His experience has been "... that there is ... little difference between lousy programs written in COBOL and lousy pro-grams written in PL/I." He concludes by stating: "But while the programming language used doesn't seem to be highly correlated with good (or lousy) programs, *the underlying design does*.[8]" As a result of structured systems design and programming discipline, I've seen more excellent programs in the last two years which run correctly the first time than I have in the previous 14 years using traditional design and coding techniques."

The conclusion of the above is that the *design process* (or *structured system design* is the important point to be continually emphasized. Lose sight of that, and the language used doesn't really make that much difference!

Premise 4: When programmers use the so-called "structured" programming languages (e.g., PL/I, ALGOL, PASCAL), they are susceptible to falling into the trap of thinking that the mere use itself of these languages will effect beautifully understandable and "correct" code. This has been shown most emphatically to not be the case; they have simply been lulled into *forgetting about DESIGN!*

[8] Italics are ours.

4.2 The Excesses of the New Programing

The present "software crisis" is not new. It, or its forerunner, started some ten years ago — in the late 1960's. Since up till then people had thought programming was supposed to be easy, the reaction to this paradoxical manifestation was both alarming and excessive. Knuth notes[30] that, "as a result of the crisis, people are now beginning to renounce every feature of programing,that can be considered guilty by virtue of its association with difficulties. Not only **go to** statements are being questioned; we also hear complaints about floating-point calculations, global variables, semaphores, pointer variables, and even assignment statements."

(Knuth's next statement would be humorous, were it not based on a realistic appraisal of current events. As such, it indicates the excesses of the New Programminq.) Knuth next states: "Soon we might be restricted to only a dozen or so programs that are sufficiently simple to be allowable; then we will be almost certain that these programs cannot lead us into any trouble, but of course we won't be able to solve many problems."

4.3 The Counter-revolution

Continuing, Knuth states[26]:

> In other words, it seems that fanatical advocates of the New Programming are going overboard in their strict enforcement of morality and purity in programs. Sooner or later people are going to find out that their beautifully-structured programs are running at only half the speed — or worse — of the dirty old programs they used to write, and they will mistakenly blame the structure instead of recognizing what is probably the real culprit — the system overhead.... Then we'll have an unfortunate counter-revolution, something like the current rejection of the "New Mathematics" in reaction to its over-zealous reforms....
>
> In the mathematical case, we know what happened: The intuitionists taught the other mathematicians a great deal about deductive methods, while the other mathematicians cleaned up the classical methods and eventually 'won' the battle.

Knuth envisions that a similar thing will eventually happen in the case of computer science: "...purists will point the way to clean constructions, and others will find ways to purify their use of floating-point arithmetic, pointer variables, assignments, etc., so that these classical tools can be used with comparative safety."

To this list of "classical tools", we shall add *Standard FORTRAN*, with its use of GOTO's. We wholeheartedly concur with Knuth that former excessive misuse of a tool is insufficient grounds for the total abandonment of that tool — especially when the replacement (open to its own set of abuses) has fared equally as bad more often than not!

Our suggestion for "purifying" the misuse of "classical" FORTRAN is the introduction of the *FunctionaL Block*. With the advent of this concept, we not only retain the recognized efficiencies of *execution* and *core-allocation* inherent in standard FORTRAN, but we also

achieve *highly understandable* code. Moreover, of equally high importance, is that the process of programing with functional blocks in a language requires the use of an interactive programming aid, which — through its detection of "improper GOTO branches" — keeps directing the programmer back to the key factor: *DESIGN*. This concept puts the emphasis (*overhead*) where it's needed — in the DESIGN/coding stage — not, however, in the execution stage, as do the "structured" languages.

4.4 Efficiency Is Still Important

Some have attempted to excuse the execution overhead of their structured languages by minimizing the need for speed. Are not computers much faster than they used to be?, they query. While the answer to this is a resounding *Yes*, yet it's not exactly the argument you present to a DP manager who's facing a costly upgrade, simply because he can't meet all his scheduling — even though he's running a 24 hour shop. Even a 5% increase in efficiency would "give" him an hour more each day.

Prof. Knuth talks a lot about efficiencies. Yet, most admittedly, he warns (several times, in fact) against premature optimization[31]: "We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Having so advised, however, he then continues:

> Yet we should not pass up our opportunity in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. . . .
>
> After a programmer knows which parts of his routines are really important, a transformation like doubling up of loops will be worthwhile. Note that this transformation introduces **go to** statements - and so do several other loop optimizations; I will return to this point later.

4.5 Dijkstra and Efficiency

Prof. Dijkstra's realistic appraisal of such optimizations as Knuth discusses above may shock some of the New Programming Purists. Concerning Dr. Dijkstra's reaction, Knuth states[32]: "He went on to say that he looks forward to the day when machines are so fast that we won't be under pressure to optimize our programs; yet

> For the time being I could not agree more with your closing remarks: if the economies matter, apply 'disciplined optimization' to a nice program, the correctness of which has been established beyond reasonable doubt. Your massaging of the program text is then no longer trickery ad hoc, it is perfectly safe and sound.

"It is hard for me to express the joy that this letter[33] gave me; it was like having all my sins forgiven, since I need no longer feel guilty about my optimized programs."

4.6 P → Q Conversion

It is that old problem — a maintainable, understandable program *vs.* an efficient program. In the world of the New Programming, you apparently must choose one or the other. Coexistence, it seems, is unobtainable in the *same* program at the *same* time.

Prof Knuth presents a very interesting solution to this problem — the P → Q conversion[34]:

> A programmer should create a program P which is readily understood and well-documented, and then he should optimize it into a program Q which is very efficient. Program Q may contain **go to** statements and other low-level features, but the transformation from P to Q should be accomplished by completely reliable and well-documented 'mechanical' operations.
>
> At this point many readers will say, 'But he should only write P, and an optimizing compiler will produce Q.' To this I say, 'No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable.' I have another alternative to propose, a new class of software which will be far better.

Knuth goes on to propose an interactive type of compiler - i.e., one in which the programmer interacts with the optimizing compiler by specifying in the *source* language itself just how optimization is to proceed. Referring obviously to an extremely high level language, Knuth reflects:

> Once we have a suitable language, we will be able to have what seems to be emerging as the programming system of the future: an interactive *program-manipulation system*, analogous to the many symbol-manipulation systems which are presently undergoing extensive development and experimentation....
>
> The programmer using such a system will write his beautifully-structured, but possibly inefficient, program P; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one.... The original program P should be retained along with the transformation specifications, so that it can be properly understood and maintained as time passes.

4.7 A Workable Alternative - *Now*

What we've been proposing is very analogous to Knuth's proposal — as far as the end results to be achieved. However, instead of working in a very high level language (such as P), then interacting with a compiler to produce Q, we work in Q directly (where Q is an intermediate level language, such as standard FORTRAN), and interact with a block-renumbering, GOTO-validating programming aid to achieve an *extremely understandable* and *verifiable* module programmed *directly in Q*! Two important points should be noted:

1. both execution and documentation apply to the *same* program
2. both standard FORTRAN and our RENUMF 2.3 programming aid are available *now!*

4.8 Misuse of Automatic Structuring Machines Considered *Harmful*

Prof. Knuth espoused the P → Q transformation, utilizing an interactive compiler. Two points should be noted concerning this: (1) P is highly structured to start with, and (2) the P → Q transformation — rather than being completely automatic — requires a design-type of effort on the programmer's part as well.

The opposite transformation (i.e., a Q → P transformation) has also received much attention. (See Knuth[35] for a review of this.) As before, Q can contain GOTO's, as well as other low-level constructions, while P should be a high-level, well-structured program. Attempts at this Q → P conversion have, however, concentrated on only one aspect: elimination of GOTO's. Moreover, such *structuring machines* proceed automatically, rather than requiring programmer interaction. The result has been that they work fairly well — if Q is carefully designed (i.e., structurally organized) to *start with*.

The opposite is not true, however. If Q is poorly designed to start with, so will the transformed P version similarly be poorly structured. Both Knuth and Dijkstra concur on this point. Knuth states[23]: "If such automatic **go to** elimination procedures are applied to badly structured programs, we can expect the resulting programs to be at least as badly structured. Dijkstra pointed this out already[12] ... saying:

> The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

"In other words, we shouldn't merely remove **go to** statements because it's the fashionable thing to do; the presence or absence of **go to** statements is not really the issue. The underlying structure of the program is what counts, and we want only to avoid usages which somehow clutter up the program. Good structure can be expressed in FORTRAN or COBOL, or even in assembly language, although less clearly and with much more trouble. The real goal is to formulate our programs in such a way that they are easily understood."

Our recommendation, concerning the use of an automatic structuring machine for converting standard FORTRAN to one of the so-called *structured* FORTRAN's, is that this conversion be applied *only after* the programmer has cleaned up the code by recasting the standard FORTRAN program in the form of Functional Blocks. Then such a conversion can take place efficiently. (Of course, once the standard FORTRAN program has been *clarified* in the form of Functional Blocks, there is really no advantage to going over to a less efficient "structured" FORTRAN.)

Premise 5: Use of completely automatic structuring machines on poorly structured low-level programs, in order to convert them to jumpless "high-level" programs, accomplishes one thing only: removal of GOTO's. Unfortunately, the clarity of

these transformed programs, as well as the efficiency with which they can be maintained, *is not improved*.

Corollary: There is no substitute for the *interactive* involvement of the programmer. A programming aid such as RENUMF, which aids the programmer in the structuring process, not only interactively involves the programmer, but also significantly reduces the manpower requirements involved in the process.

4.9 A Reflection on Efficiencies and compilers — in 2006

While the "standard" FORTRAN of 1977, FORTRAN 77, has remained a steadfast and reliable workhorse to this very day, yet the so called "structured" languages have changed. Now they are the Object Orientated Programming languages, with their own level of overhead, as well as the *new* FORTRAN, the FORTRAN 90 and upward series — which, as Andrew Scriven[43] notes, "... will be nothing like Fortran and will be called Fortran".

Of particular note with respect to efficiency degradation in the new "FORTRAN" are two usages,

- the `module` statement[9]
- user derived types[10]

which the NASA Langley Research Center publication by William Kleb *et al*[47] has commented on in some detail:

> ... use of the `module` construct can incur severe penalties for unformatted disk I/O. The `module` interface is over thirty times slower than the data transferred via a conventional argument list on an SGI.
>
> As in the case of `modules`, it was found that the use of derived types can also incur severe execution penalties ... this coding idiom can yield execution times more than thirty times slower for unformatted disk I/O and nearly a factor of three slower for floating-point operations over the argument list model.

We feel that Andrew Scriven's[48] comment succinctly summarizes the underlying problem: "When you move further away from the underlying machine instructions, and increase the level of abstraction, you make the user's job easier and the compiler writer's job harder."

And therein is the key — the compiler. As an example, in a recent personal communication Dr. William L. Kleb[49] explained to us that the above cited inefficiencies with respect to usage of the `module` and derived types were due to the particular compiler which they were using on their SGI hardware. When they went to a different compiler, they were able to obtain efficiencies the same as, *if not better than*, those under the former FORTRAN 77 programs.

The compiler is the key.

[9] Here we use the FORTRAN 90 definition of this term.
[10] These are similar to the `STRUCTURE` construct within VAX FORTRAN 77.

5 A Critique of Functional Blocks

5.1 Not Novel With Us

The concept of a *block of code* is certainly not novel with us.[11]. It is simply the decomposition of a task as a whole into a simpler set of understandable parts. Knuth states[21]: "We understand complex things by systematically breaking them into successively simpler parts and understanding how the parts fit together locally."

Blocks of code, admittedly, are not all that novel. Two of us have been breaking our modules of code (be they in FORTRAN or in Assembler) into blocks since 1963. Short of that, however, we look back on some of our old code and see that our *blocks* usually contained a composite of *several* functions. Moreover, we thought nothing of branching out of one block into the midst of another block.

A newer concept of blocks, or *chunks*, has been emerging since then, however. Robert E. Horn, for example, who has developed a systematic method (*Information mapping*) of organizing documentary material,[36] refers to chunks as "Information Blocks". "One rule", he states, "is to put only information of *one* functional kind in an information block."

David Frost states[37] that "...chunks based on function are important for handling procedures. A procedure's purpose is to do things (i.e., perform functions). A good functional chunk is an entity that (1) does one thing, (2) can be named, and (3) its function can be described easily in one sentence without resorting to a great many if's, and's, and but's."

To the above contributions we need add only Knut Bulow's thoughts on "Programming in Book Format"[38] to complete the then-existing picture of a functional block. Mr. Bulow advances four points:

1. The blocks (which he terms *chapters*) should be numbered (just as chapters in a book are numbered)
2. No more than a single line of comment should precede each block of code
3. If more than a single line of comment is required to describe an algorithm, etc., in certain blocks, then these more extensive comments should be placed *at the beginning* of the module, just as a table-of-contents is placed at the beginning of a book
4. Each chapter should "...be as self-sufficient as possible, with limited GO TO's branching to other chapters"

5.2 Our Concept: Renumbering by Blocks

To the framework of a functional block as formulated by Horn, Frost, and Bulow, it occurred to us how meaningful it would be if the statement numbers in each block started *with the number of that block*. This one additional characterization of functional blocks would now allow programmers to associate statement numbers *with the functional aspects of their code*.

[11] However, our concept of "renumbering by functional blocks" and of checking for "improper GOTO branches", *did originate with us*

Additionally, as we illustrate in our companion technical report, *Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)*, and as can be seen from the example in the Appendix, the *significant* or *leading* portion of each statement number is equivalent to the outline-number in the respective outline of the *processes*. For example, if *second*-level blocking is utilized - i.e., where each major block can in turn be divided into sub-blocks - then the first *two* digits of the statement numbers are significant, so that the set of numbers (4200, 4210, 4220, ...) would be associated with the process of sub-block 4.2.

Moreover, we feel our concept of "renumbering by functional blocks" *has removed* most of Prof. Knuth's objection to numeric labels[25]:

> We've already mentioned that **go to**'s do not have a syntactic structure that the eye can grasp automatically; but in this respect they are no worse off than variables and other identifiers. When these are given a meaningful name corresponding to the abstraction (N.B. *not* a numeric label!), we need not apologize for the lack of syntactic structure.

5.3 Structuring By Information Blocks

Prof. Knuth stated above that GOTO's lack a syntactic structure that the eye can grasp. We rectify that by viewing each functional block (or sub-block) as an *information block*, separated from the other information blocks by appropriate *horizontal Lines*, as required by Horn[36] in his "Information Mapping". In other words, we utilize *horizontal indentation*, rather than the *vertical indentation* of syntactic structuring. We feel that this method effects *just as meaningful a structure*, which the eye can grasp automatically, as that claimed for syntactic structuring.

A given functional block can obviously be decomposed into sub-blocks. If we refer to a major functional block as a Level-1 block, then we can refer to the sub-blocks composing it as Level-2 blocks (or sub-blocks). Similarly, if a given Level-2 sub-block is in turn further decomposed into sub-blocks, then we can refer to these as Level-3 blocks (or sub-blocks).

What we have just described above is referred to as *hierarchical decomposition*. Concerning such a hierarchy of various-level blocks, it is important to emphasize that the *type* of horizontal line(s) preceding each level of block in the hierarchy can *very visually* identify the level of the block. Moreover, RENUMF is designed such that the choice is completely at the disposition of the user.

We ourselves (please refer to the Appendix for an illustration) like to use a double-row of asterisks (from cols. 2–72) to denote Level-1 blocks, while a double-row of dashes (from cols. 7–72) denote Level-2 blocks. The single line of comment preceding each block is imbedded between the two respective lines. Moreover, in the case of Level-1 blocks, the BLOCK-ID number is also included between the two lines of asterisks at the very beginning. For a Level-3 block, we use a single row of equal signs (from cols. 9–11), with the comment on the same line, starting in column 12.

RENUMF offers two different ways in which the user can identify both the beginning of blocks as well as their level. The *Block Number Descriptor* permits the user to define the *number* of a block (or sub-block), while the *Header Structure Descriptor* identifies the beginning (and level) of a block by the *structure* of the horizontal line(s) preceding it.

5.4 Single Line of Comment

Prof. Knuth has a very valid warning concerning the use of comments[39]:

> Accompanying comments explain the program and relate it to the global structure illustrated in flow charts, but it is not so easy to understand what is going on; and it is easy to make mistakes, partly because we rely so much on comments which might possibly be inaccurate descriptions of what the program really does.

We propose two stipulations regarding the use of comments which we feel completely rectifies his objections:

1. *No more* than a single line of comment should precede any given-level block;
2. The comment should state the function, or process, to be performed in the given-level block; i.e., it should state *what* is to be done, but not *how* it is to be done.

We now elaborate on each of these points. Concerning the first, two benefits are to be achieved:

1a. The code itself, as well as the structure achieved through horizontal indentation, are not cluttered up by *excessive* comments

1b. Because there is only a *single* line of comment preceding each block, the chance of the comment being erroneous is significantly reduced.

Moreover, since programing in Functional Blocks is functionally-orientated, the programmer of necessity *must* view the single comment line as though it were a part of the code. As a result, it is virtually impossible for a comment line to be incorrect.

The second stipulation is given for two reasons:

2a. Since programing in Functional Blocks *is* functionally-orientated, it is very necessary to state the process, or function, to be performed in each given-level block

2b. The function to be performed (i.e., *what* is to be done) in any level block is *less subject to change* than *how* it is to be done.

This second reason (#2b) is especially true in a Level-1 major block, where a change in algorithm does not reflect a change in the process to be performed, but only how the process is to be performed. (Admittedly, however, a change in algorithm at the major block level will probably cause all sub-level blocks to be completely changed, including their preceding comment lines.)

5.5 HIPO-Chart Processes

While on the subject of comments, it is of interest to note that the comments preceding the Level-1 blocks are precisely the *processes* of the related HIPO chart. If comments preceding sub-blocks are included as well, with appropriate vertical indentation and numbering, then the complete *outline* of the processes to be accomplished becomes available.

5.6 Proper GOTO Branches: Verifiable Code

Premise 2 states: "The GOTO construct is allowed in program proofs if, for each label L, there is a logical assertion a(L) which is to be uniquely true, *for each appearance of the label L*, whenever we reach L."

Our concept of "proper GOTO branches" is a necessary, although not sufficient, condition for Premise 2 to be true. To prove it is a necessary condition, we need only show that it is not possible[12] for Premise 2 to be true when "*im*proper GOTO branches" exist:

> If, at the beginning of Block (or Sub-block) #L, marked by label L,[13] we assume a logical assertion α(L) to be true, then a branch (from an external block) to Block #L, *other than to its beginning* (say to label L+ϵ), must of necessity presume some other logical assertion β(L+ϵ) to be true, simply because the *intervening code*[12] between labels L and L+ϵ modifies the assertion α(L) at L to a new assertion β(L+ϵ) at L+ϵ.

From a different perspective, we realize that a programmer, who branches into the *midst* of a block, is viewing that block in a *microscopic* sense (as a detailed set of *parts*) rather than in a *macroscopic* sense (as a *single functional entity*). The whole concept behind Functional Blocks, of course, is that each block *must be viewed* as a single functional entity. Thus, from the standpoint of *other* blocks, the interior contents of that block must be *hidden* from them — in fact, does not even concern them. The utilization of this *hiding principle* (which is *just another way* of defining "proper GOTO branches") is what insures that any maintenance changes made to a given block will be *isolated* to that block, rather than rippling over to other blocks.

Even when a module is coded in functional blocks, however, and accordingly contains *only* "proper GOTO branches", this is still not a sufficient guarantee *in itself* that every reference to a label L is with respect to the *same* logical assertion, α(L). The point we must make here is that is is not sufficient just to specify the unique function that a block is to perform. For example, the `COS` function is a unique function; yet it can certainly be called under a variety of different conditions. Similarly, a given functional block could conceivably be referenced in certain cases under the assumption that assertion α(L) was true, and in other cases under the assumption that assertion β(L) was true.

[12] A more proper wording should be "not *necessarily* possible", because trivial cases could exist where the intervening statements between the labels L and L+ϵ could be "NOP" statements, such as `CONTINUE` or `GO TO L+`ϵ. Excluding such trivial cases however, the stronger wording of "not possible" is admissible.

[13] In reality, the value of the label will be L•a, where **a** is defined by the **aMODb** in the appropriate BLOCKID descriptor. This does not diminish the proof, however.

For the predicament presented above, several different solutions are available. One is to have two functional blocks, each performing the same function, but with one at label L, and the other at label M. Then Block #L is branched to only when α(L) is true, while Block #M is branched to only when α(M) is true. This approach is wasteful of memory, of course. A preferred method in this case would be either to transform the functional block into an in-line subroutine, or to make it into a standard external subroutine. In either case, the function,is then "called" under the appropriate logical condition.

5.7 A More Precise, Description Preceding a Functional Block

We previously stated that the single line of comment preceding a given-level functional block should describe the process to be performed by that block. In view of the above discussion, however, we propose that the comment, up to now consisting of just a single process-description, e.g.,

C *process-description*

be replaced by a two-part comment, the first part being the logical assertion, or *event*, that is to be true when we reach that block, and the second part being the process to be performed, e.g.,

c <*event*>: *process-description*

For purposes of visual emphasis, we enclose the *event* in the <> delimiters, and separate the two with a *colon*. The two must still occupy only a single line, of course.

We recommend this *even when all events are unique*. The reason is quite simple: it *emphasizes* to the programmer exactly *what* is happening. (Note: this is a *DESIGN* concept.) We quite vividly illustrate this in our companion technical report, *Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)*, as well as in the example shown in the Appendix of this paper.

5.8 Self-Documenting

Since the single-line comments preceding the functional blocks are identical to (or equivalent abbreviations of) the *processes* of the module's INPUT-PROCESS-OUTPUT diagram, essentially the only documentation that therefore need be maintained on the module is the module-code itself! Hitherto, it has been thought that a module was not documented if *external* documentation concerning the module was not available. (Interestingly, by this definition, many modules were not documented.) And even when external documentation was provided, it was usually not updated to reflect changes made to the module. Hence a paradoxical situation existed: because the module code was unstructured, it definitely needed external documentation; yet this external documentation was itself unreliable.

Modules coded in Functional Blocks, however, provide their own documentation. When maintenance is performed on the module, the programmer knows that the "module documentation" is being simultaneously updated along with the module itself, since they are one-and-the-same!

Our concept of *external* documentation is that it should be extremely insensitive (i.e., invariant) to maintenance and enhancement. Stated another way, we are saying that external documentation should be restricted to that class of documentation which is (nearly) invariant to maintenance and enhancement. Now the traditional method of documentation, in which the module code was augmented with an external description of how the module accomplishes its processes, was unfortunately *very sensitive* to module changes.

There are, however, several very invaluable external types of documentation that can be kept on a module, in addition to the module-code itself, which are fortunately quite insensitive to module changes. These are:

1. A description of what the module does
2. A description of each variable in its argument list (if present), describing
   (a) variable type
   (b) whether it is Input, Output, or both
3. A description of labelled COMMON's, if any, employed
4. An alphabetical list of the variables utilized in the module, with each of their —
   (a) mnemonics spelled-out
   (b) types specified
   (c) purposes briefly described

These can, if desired, be maintained in a "Prologue Section" at the very beginning of the module. Documentation of this type, besides being very useful, is extremely insensitive to changes made to the module.

In conclusion, documentation of this kind, along with the module-code itself, is transferable to any other of your programmers with a *maximum of understandability*. From this standpoint, there is an extremely important *additional* dividend: the impact of programmer-turnover on the transferability of your code is significantly reduced, if not completely eliminated!

5.9 How Many Blocks in a Module?

This question is not necessarily equivalent to the related question, "How long should a module be?" From the standpoint of programing in a syntactically structured language, however, this latter question is the relevant one, since a solid block of nested IF THEN ELSE's and DO WHILE'S, etc., becomes very difficult to comprehend if the module length is not restricted. This is why Harlan Mills introduced the concept of a *segment*[40] of code, in which the length of the module (or segment) is restricted to the number of lines of program text that will fit on a single page (i.e., 50 or fewer lines of code).

Modules coded in standard FORTRAN, utilizing Functional Blocks, do not face this similar restriction. Interestingly, the restricting "length" of a module coded in functional blocks is dependent upon the *number* of functional blocks, and the *length* of each functional block, rather than upon the total module length *per se*. Even the "permissible" length of each functional block itself is dependent upon whether it is also divided into sub-blocks or not.

What, then, is the maximum number of Functional Blocks we can have in a module while still maintaining our comprehension and understandability of the module? David Frost, in his article "Psychology and Program Design"[37], answers this for us:

> The number of pieces of information (chunks) we can think about at any one instant has been estimated to be approximately seven (7±2) and more recently to be between five and seven. The number 7±2 crops up in so many psychological experiments that it has been called a "magical number".
>
> ...For example, a telephone number is much more easily comprehended and remembered as (602) 555 2342 X441 (four chunks) rather than as 6025552342441 (one big chunk with thirteen units in it).

From this standpoint alone, we accordingly suggest that the maximum number of Functional Blocks in a single module be 7±2. (It is of interest to note that we, independently of Frost's information, had from other considerations concluded that *9* should be the maximum number of blocks.)

What if we find that more than nine blocks are involved in a single module? Quite simply, that is strongly indicative that we should decompose that module into two or more separate modules.

Ok, now what about the suggested maximum length of each block? Here, based on our own experience, we come up with a suggested maximum length of 15→20 lines of code. Now this is applicable to the length of the block only if it is in fact just a single block, i.e., not in turn divided into sub-blocks. If we divide that single block into sub-blocks (the number of which is not to exceed 7±2), then each of these sub-blocks can in turn have a maximum length of 15→20 statements. See how we've increased the effective length of a major Functional Block *while still maintaining a high degree of comprehension*. This is possible, of course, only because our blocks are *Functional Blocks* in the first place (i.e., in which *only* "proper GOTO branches" are utilized).

Conceptually, the same procedure could be repeated for any given Level-2 sub-block, dividing it into Level-3 sub-blocks (not to exceed 7±2 in number). We have found from experience however, that our comprehension does begin to degrade *slightly* when we go to Level-3 sub-blocks. And when we go to Level-4 sub-blocks, the degradation becomes greater yet.

What if we find that Level-4, or lower, decomposition is required in a given functional block? Quite simply, that is *very* strongly indicative that we should decompose that module into two or more separate modules.

5.10 Can there be an Exception to *Total* Module Length?

Of course there can be exceptions. One notable exception occurs when we structure a module for a paying client. If the module is overly long, with obvious breaking points apparent, we can suggest it to the client; but in the final run of things that is something that the client must elect to do. We never take it upon ourselves to do that.

6 Conclusion

6.1 Functional Blocks offer a powerful alternative

Programing in Functional Blocks offers a powerful alternative to programming in a syntactically structured language, such as the Object-Orientated languages, or even the new FORTRAN 90 series. The understandability, verifiability, and maintainability of the resulting code is equal to, if not better than, that of the above-described syntactically structured code! Moreover, the danger of being lulled into forgetting about the underlying DESIGN considerations, to which the syntactically structured languages are partial, essentially does not exist in code utilizing Functional Blocks, since the programmer must be interactively involved (utilizing a programming aid such as RENUMF) to make the procedure work.

The single-line comments preceding each major Functional Block are the *processes* of the accompanying HI*P*O diagram for the module. The code itself (with accompanying single-line comments) serves as the documentation for the module.

The maximum length of a module coded in Functional Blocks is dependent upon the total number of Functional Blocks present, and their lengths, rather than upon the total number of lines of code *per se*.

The concept of coding in Functional Blocks is directly applicable to standard FORTRAN, with the result that the *high efficiency* of standard FORTRAN, such as FORTRAN 77, can be utilized in producing code which is at the same time highly understandable, verifiable, and therefore readily maintainable.

6.2 Converting legacy FORTRAN to FORTRAN 90

Because many scientific FORTRAN 77 subroutines are both very straightforward and very short in length, the use of a F77 to F90 converter on such modules accordingly works very well.

But not all scientific legacy FORTRAN programs or subroutines, as well as most business legacy FORTRAN subroutines, are all that straightforward nor all that short. And, so it seems, the longer these legacy code units are, the more poorly structured and overly complex they also are. On these rather complex code units, attempts at applying automatic F77 to F90 converters fare quite poorly, as Dijkstra himself had pointed out[12].

Our suggestion is to first structure these poorly structured modules by our methodology of Functional Blocks, by which — in a rapidly converging iterative process — a well structured and highly understandable module is soon arrived at. *Now* use your F77 to F90 converter, and all will be well.

Characterization of a Functional Block

A Functional Block is a block of code possessing at least the following eleven characteristics:

1. It performs a single function.
2. It exists at a specified Hierarchical *Level*. (The major blocks required to accomplish the module-task are Level-1 blocks; a decomposition of any major block into sub-blocks results in Level-2 blocks; etc.)
3. It is preceded by horizontal line(s) which
   - denote its level,
   - contain a single line of comment,
   - contain an optional block number.
4. Its single line of comment preceding it is of the form,

   <*event*>: *process-description*

   where *event* denotes the logical assertion that is uniquely true upon entry into the block, and *process-description* describes the process to be performed.
5. If it is a major (Level-1) block, then the block number is mandatory. (While explicit numbering is optional for sub-blocks, yet every sub-block has at least an implicit block number associated with It.)
6. Statement number labels for blocks should be chosen so that the significant portion of each statement number is equal to a suitable multiple of the block number.
7. Ordering of statement number labels within a given block occurs in the non-significant portion of the statement numbers, and is sequentially ascending by a selected increment.
8. The value of the non-significant portion of the *first* statement number in the block is zero if the statement number label occurs on the *first* executable statement within the block; otherwise it is set to the non-zero increment value.

   Note: Once the horizontal *descriptor* lines specified in Step #3 are in place, then RENUMF *automatically* performs the block-renumbering as stipulated above in Steps #6, #7, and #8.
9. The only permissible GOTO branch from any *external* major (Level-1) block to a given major block is to its *beginning*. This is termed a "proper GOTO branch".
10. The same rule applies to all Level-2 sub-blocks within the domain of a given major (Level-1) block; etc.

    Note: RENUMF automatically reports those statement numbers which violate Steps #9 and #10 above, as well as stating the *level* of the higher hierarchy block involved.
11. Only statement number labels which are referenced are retained. *Un*referenced labels tend to "clutter up" the code and confuse the programmer's grasp of the flow of processing.

An Example

Some background

The example shown in Figures 1.a, 1.b, and 1.c employs an algorithm which is essentially identical to the one employed by Dr. Peter G. Anderson[42], presently Professor Emeritus in the Computer Science Department of the Rochester Institute of Technology, back when he was instructing a FORTRAN class to beginning programmers in 1976 at the New Jersey Institute of Technology.[14]

This algorithm is ideally suited to our illustration of Functional Blocks, because it is short, not *overly* complex, and yet certainly *not trivial*. It readily illustrates how Functional Blocks permit an otherwise complex whole to be *clarified* through decomposition into a set of readily understandable parts. Moreover, most all aspects of Functional Blocks described in the previous *characterization* are illustrated in this example.

The example involves the CHECKER subroutine,[15] which places checkers on the NxN checkerboard such that the four corners of all possible squares that can be formed do not contain the same colored checkers. The algorithm employed puts checkers on the board, one at a time. If it runs into a blocked situation where neither colored checker will work, it backtracks (*i.e.*, retreats), changing previous decisions.

Please note the following three points regarding the code shown in these figures:

1. What is actually shown in Figures 1.a and 1.b is the printout of the RENUMF processing of the subroutine version of Dr. Anderson's CHECKERBRD program, while Figure 1.c contains two of the three associated cross references, along with the Functional Block Outline, all generated by RENUMF
2. While columns 73–80 of the *printout* of this code contains an ID and a sequence number, yet these same columns within the code *source file* output by RENUMF are left blank (these are the default options used by RENUMF)
3. Some blank "spacer lines" (i.e., completely blank lines with only a "C" or an "*" in column 1) have been removed in the source listings in Figures 1.a and 1.b, in order to make these fit onto a single page

[14] The authors not only acknowledge Prof. Anderson as the source of the algorithm employed in the example, but also wish to point out that Dr. Anderson employed an interesting alternate approach to structuring standard FORTRAN, in which the various parts of the program are clearly evident through the use of horizontal indentation to achieve a vertical alignment encompassing each program part, as illustrated on pages 10–12 of his brochure, *Structured Programming: Style Manual for FORTRAN Programmers*.

In the process of revising our report, originally completed in 1977, we found that we had lost the copy of Dr. Anderson's style manual, which we desired to have in order to double check previous statements. We accessed him through his web site, **http://www.CS.RIT.edu/~PGA**, and requested another copy, which he graciously sent us. It also afforded us the chance to visit with him, recalling those earlier days, some 30 years ago, when we first had the privilege of corresponding with him regarding the structuring of FORTRAN. (If you wish to contact Dr. Anderson through e-mail, it is advisable to call him first, stating that you are going to e-mail him; otherwise your e-mail will not make it through his filter.)

Unlike us, Dr. Anderson has moved on from FORTRAN to a newer language. What is that newer programming language? Go to his web site to find out.

[15] We extracted the non-trivial part of Dr. Anderson's CHECKERBRD program into the CHECKER subroutine, with arguments `(N, BOX, NMRET, IERROR)`, where `N`, the NxN board size, is input from the main CHECKERBRD program, and `BOX`, `NMRET`, and `IERROR` are output to it, where `BOX` is the NxN array of checkerboard color values, `NMRET` is the number of retreats, and `IERROR` is an integer error code.

Figure 1.a: Listing of subroutine version of Dr. Anderson's CHECKERBRD program (Part 1 of 2)

```
    Directory: M:\GOTOs_OK                                        Input file: checker.for
    Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)           RENUMF 2.3.0, Oct. 25, 2004
    LISTING OF SOURCE DECK "CHECKER                    "          04/28/06          15:57:31          PAGE   1

1       SUBROUTINE CHECKER (N, BOX, NMRET, IERROR)                                  CHEC0010
    *                                                                               CHEC0020
    * Abstract ***[04/28/2006]*********************************************CHEC0030
    *   Places checkers on NxN board so that no corners have all same color.CHEC0040
    *                                                                               CHEC0050
    * Keywords                                                                      CHEC0060
    *   checkers, corners of squares. retreat                                       CHEC0070
    *                                                                               CHEC0080
    * Purpose                                                                       CHEC0090
    *   The CHECKER subroutine places red and black checkers on a NxN board CHEC0100
    *     so that the corners of all squares that can be formed do not have CHEC0110
    *     checkers of the same color.                                               CHEC0120
    *                                                                               CHEC0130
    * Arguments                                                                     CHEC0140
    *   N       INPUT         INTEGER           SCALER                            CHEC0160
    *     The size of the NxN checkerboard, where 2 <= N <=20.                      CHEC0170
    *                                                                               CHEC0180
    *   BOX     OUTPUT        INTEGER           ARRAY                             CHEC0190
    *     Array of checkerboard color values, where 0=RED and 1=BLACK,              CHEC0200
    *                                                                               CHEC0210
    *   NMRET   OUTPUT        INTEGER           SCALAR                            CHEC0220
    *     Total NuMber of RETreats required for the NxN checkerboard.               CHEC0230
    *                                                                               CHEC0240
    *   IERROR  OUTPUT        INTEGER           SCALAR                            CHEC0250
    *     Integer ERROR value returned:                                             CHEC0260
    *          0 = valid return                                                     CHEC0270
    *          2 = invalid input value of the N input arg                           CHEC0280
    *                                                                               CHEC0290
    * Commons loaded:                                                               CHEC0300
    *   None                                                                        CHEC0320
    *                                                                               CHEC0330
    * Commons used:                                                                 CHEC0340
    *   None                                                                        CHEC0360
    *                                                                               CHEC0370
    * Modules called: ADVANC, MIN0,   RETRET                                        CHEC0380
    *                                                                               CHEC0390
    * Databases (associated data sets), or Files accessed:                          CHEC0400
    *   Data base: None                                                             CHEC0410
    *   Files:     None                                                             CHEC0420
    *                                                                               CHEC0430
    * Errors                                                                        CHEC0440
    *   IERROR = 0, valid return                                                    CHEC0450
    *   IERROR = 2, invalid input value of N                                        CHEC0460
    *                                                                               CHEC0470
    * Notes                                                                         CHEC0480
    *   None.                                                                       CHEC0500
    *                                                                               CHEC0510
    * History                                                                       CHEC0520
    *     [Key: PGA = Dr. Peter Gordon Anderson,                                    CHEC0530
    *           RCW = Ronald C. Wackwitz]                                           CHEC0540
    *   Original , 09/01/1976, #01, PGA, Initial release as ENTIRE program. CHEC0550
    *   CHECKER  , 05/31/1977, #02, RCW, Extracted checker-putting portion  CHEC0560
    *                                    of Dr. Anderson's program, cast it CHEC0570
    *                                    into the form of Functional Blocks,CHEC0580
    *                                    and added this header and          CHEC0590
    *                                    documented local variables.        CHEC0600
    *   CHECKER  , 04/28/2006, #03, RCW, Ran fresh RENUMF run on CHECKER.forCHEC0610
    *                                                                               CHEC0620
    * End ***************************************************************CHEC0630
    C                                                                               CHEC0640
    C     ------------------------------------------------------------------CHEC0660
    C     Local Variables.                                                          CHEC0670
    C     ------------------------------------------------------------------CHEC0680
    C      Name    Size  Type  Description                                          CHEC0690
    C     ------  ------ ----- ------------------------------------------CHEC0700
    C     ADVANC         Sub   ADVANCe current (I,J) to next square         CHEC0710
    C     BLACK          I     Integer value set =1 to denote BLACK color   CHEC0720
    C     I              I     Index used for rows    within BOX array      CHEC0730
    C     J              I     Index used for columns within BOX array      CHEC0740
    C     K              I     Index used to search all MINNY squares       CHEC0750
    C     MIN0           I     Integer function returns MIN of arg values   CHEC0760
    C     MINNY          I     Max number of squares within current (I,J)   CHEC0770
    C     OK             L     Flag denoting whether all squares are OK     CHEC0780
    C     RED            I     Integer value set =0 to denote RED   color   CHEC0790
    C     RETRET         Sub   RETREaT from current (I,J) to a RED square   CHEC0800
    C     SUM            I     SUM of color values within current square    CHEC0810
    C                                                                         CHEC0830
    C                                                                               CHEC0840
    C*******************************************************************CHEC0850
    C*****Declarations.                                                  *CHEC0860
    C*******************************************************************CHEC0870
    C                                                                               CHEC0880
    C     ------------------------------------------------------------------CHEC0900
    C     Intrinsics.                                                               CHEC0910
    C     ------------------------------------------------------------------CHEC0920
2       IMPLICIT NONE                                                               CHEC0930
    C                                                                               CHEC0940
    C                                                                               CHEC0950
    C     ------------------------------------------------------------------CHEC0960
    C     Arguments.                                                                CHEC0970
    C     ------------------------------------------------------------------CHEC0980
3       DIMENSION BOX(20,20)                                                        CHEC0990
4       INTEGER N,      BOX,   NMRET,  IERROR                                       CHEC1000
    C                                                                               CHEC1010
    C                                                                               CHEC1020
    C     ------------------------------------------------------------------CHEC1030
    C     Local.                                                                    CHEC1040
    C     ------------------------------------------------------------------CHEC1050
5       INTEGER SUM,    RED,    BLACK,  I,      J,      K,      MINNY     CHEC1060
6       LOGICAL OK                                                                  CHEC1070
7       DATA RED, BLACK / 0, 1 /                                                    CHEC1080
```

Figure 1.b: Listing of subroutine version of Dr. Anderson's CHECKERBRD program (Part 2 of 2)

```
     Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)             RENUMF 2.3.0, Oct. 25, 2004
     LISTING OF SOURCE DECK "CHECKER                      "          04/28/06              15:57:31             PAGE   3

     C~                                                                                                      CHEC1090
     C                                                                                                       CHEC1100
     C***********************************************************************                              CHEC1110
     C(01) <Entry>: Perform initialization.                                                                 *CHEC1120
     C***********************************************************************                              CHEC1130
  8          CONTINUE                                                                                       CHEC1140
     C                                                                                                       CHEC1150
     C     ------------------------------------------------------------------                               CHEC1170
     C     <Entry>: Zero IERROR; validate N.  If invalid, set IERROR=2 & rtrnCHEC1180
     C     ------------------------------------------------------------------                               CHEC1190
  9          IERROR = 0                                                                                     CHEC1200
 10          IF ((N.GT.1) .AND. (N.LE. 20)) GO TO 1200                                                      CHEC1210
 11          IERROR = 2                                                                                     CHEC1220
 12          RETURN                                                                                         CHEC1230
     C                                                                                                       CHEC1240
     C     ------------------------------------------------------------------                               CHEC1260
     C     <N valid>: Zero NMRET.                                                                           CHEC1270
     C     ------------------------------------------------------------------                               CHEC1280
 13    1200 NMRET = 0                                                                                       CHEC1290
     C                                                                                                       CHEC1300
     C     ------------------------------------------------------------------                               CHEC1320
     C     <Argument processing completed>: Init I & J row-column counters.  CHEC1330
     C     ------------------------------------------------------------------                               CHEC1340
 14          I = 0                                                                                          CHEC1350
 15          J = 0                                                                                          CHEC1360
     C                                                                                                       CHEC1370
     C                                                                                                       CHEC1380
     C***********************************************************************                              CHEC1390
     C(02) <(0<=I<=N) & (0<=J<=N)>: Advance one square & set it to RED; drop*CHEC1400
     C***********************************************************************                              CHEC1410
 16    2000 CALL ADVANC (I, J, N)                                                                           CHEC1420
 17          BOX(I,J) = RED                                                                                 CHEC1430
     C                                                                                                       CHEC1440
     C                                                                                                       CHEC1450
     C***********************************************************************                              CHEC1460
     C(03) <Counter now at I,J>: Test corners of all squares; skip ->#5 if OKCHEC1470
     C***********************************************************************                              CHEC1480
 18    3000 CONTINUE                                                                                        CHEC1490
     C                                                                                                       CHEC1500
     C                                                                                                       CHEC1510
     C     ------------------------------------------------------------------                               CHEC1520
     C     <Entry>: Init OK=True for current testing of squares upto (I,J).  CHEC1530
     C     ------------------------------------------------------------------                               CHEC1540
 19          OK = .TRUE.                                                                                    CHEC1550
     C                                                                                                       CHEC1560
     C     ------------------------------------------------------------------                               CHEC1580
     C     <OK=True>: Max No. squares is one less min of (I,J); save in MINNYCHEC1590
     C     ------------------------------------------------------------------                               CHEC1600
 20          MINNY = MIN0(I,J) - 1                                                                          CHEC1610
     C                                                                                                       CHEC1620
     C     ------------------------------------------------------------------                               CHEC1640
     C     <MINNY defined>: If zero, goback ->#2 to advance one more square. CHEC1650
     C     ------------------------------------------------------------------                               CHEC1660
 21          IF (MINNY .EQ. 0) GO TO 2000                                                                   CHEC1670
     C                                                                                                       CHEC1680
     C     ------------------------------------------------------------------                               CHEC1700
     C     <MINNY squares within (I,J)>: Check on colors of all their cornersCHEC1710
     C     ------------------------------------------------------------------                               CHEC1720
 22          DO 3410 K = 1, MINNY                                                                           CHEC1730
 23             SUM = BOX(I,J) + BOX(I-K,J) + BOX(I,J-K) + BOX(I-K,J-K)                                     CHEC1740
 24             OK  = OK .AND. SUM.NE.0 .AND. SUM.NE.4                                                      CHEC1750
 25    3410 CONTINUE                                                                                        CHEC1760
     C                                                                                                       CHEC1770
     C     ------------------------------------------------------------------                               CHEC1790
     C     <OK will be False if corners of any square same>: But if OK, ->#5.CHEC1800
     C     ------------------------------------------------------------------                               CHEC1810
 26          IF (OK) GO TO 5000                                                                             CHEC1820
     C                                                                                                       CHEC1830
     C                                                                                                       CHEC1840
     C***********************************************************************                              CHEC1850
     C(04) <Corners of some square same color>: Interchange & ->#3 to re-testCHEC1860
     C***********************************************************************                              CHEC1870
 27          CONTINUE                                                                                       CHEC1880
     C                                                                                                       CHEC1890
     C     ------------------------------------------------------------------                               CHEC1910
     C     <Entry>: If current (I,J) square not RED, retreat until it is.    CHEC1920
     C     ------------------------------------------------------------------                               CHEC1930
 28    4100 IF (BOX(I,J) .EQ. RED) GO TO 4200                                                               CHEC1940
 29          CALL RETRET (I, J, N)                                                                          CHEC1950
 30          NMRET = NMRET + 1                                                                              CHEC1960
 31          GO TO 4100                                                                                     CHEC1970
     C                                                                                                       CHEC1990
     C     ------------------------------------------------------------------                               CHEC2000
     C     <Current (I,J) square now RED>: Set it BLACK & ->#3 to re-test.   CHEC2010
     C     ------------------------------------------------------------------                               CHEC2020
 32    4200 BOX(I,J) = BLACK                                                                                CHEC2030
 33          GO TO 3000                                                                                     CHEC2040
     C                                                                                                       CHEC2050
     C                                                                                                       CHEC2060
     C***********************************************************************                              CHEC2070
     C(05) <Not same color>: If any squares left, goback ->#2 to advance.   *CHEC2080
     C***********************************************************************                              CHEC2090
 34    5000 IF ((I.LT.N) .OR. (J.LT.N)) GO TO 2000                                                          CHEC2100
     C                                                                                                       CHEC2110
     C                                                                                                       CHEC2120
     C***********************************************************************                              CHEC2130
     C(06) <All squares validly filled>: So return.                                                         *CHEC2140
     C***********************************************************************                              CHEC2150
 35          RETURN                                                                                         CHEC2160
 36          END                                                                                            CHEC2170
```

Figure 1.c: Cross references and Functional Block Outline for CHECKER subroutine

```
     Directory: M:\GOTOs_OK                                          Input file: checker.for
     Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)             RENUMF 2.3.0, Oct. 25, 2004
     CROSS-REFERENCE OF VARIABLES    04/28/06 15:57:31               PAGE   6
       IN SOURCE DECK "CHECKER                    "

       VARIABLE INFO         REFERENCES

       ADVANC   SUB            16:R
       BLACK    S     I         5:D     7:S    32:R
       BOX      A     I         1:A     3:D     4:D    17:S    23:R    23:R    23:R    23:R    28:R    32:S
       CHECKER  SUB            1:D
       I        S     I         5:D    14:S    16:A    17:R    20:A    23:R    23:R    23:R    23:R    28:R    29:A
                                       32:R    34:R
       IERROR   S     I         1:A     4:D     9:S    11:S
       J        S     I         5:D    15:S    16:A    17:R    20:A    23:R    23:R    23:R    23:R    28:R    29:A
                                       32:R    34:R
       K        S     I         5:D    22:S    23:R    23:R    23:R    23:R
       MINO     FUN   I        20:R
       MINNY    S     I         5:D    20:S    21:R    22:R
       N        S     I         1:A     4:D    10:R    10:R    16:A    29:A    34:R    34:R
       NMRET    S     I         1:A     4:D    13:S    30:S    30:R
       OK       S     L         6:D    19:S    24:S    24:R    26:R
       RED      S     I         5:D     7:S    17:R    28:R
       RETRET   SUB            29:R
       SUM      S     I         5:D    23:S    24:R    24:R


     Directory: M:\GOTOs_OK                                          Input file: checker.for
     Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)             RENUMF 2.3.0, Oct. 25, 2004
     CROSS-REFERENCE OF STATEMENT NUMBERS   04/28/06   15:57:31     PAGE   7
       IN SOURCE DECK "CHECKER                    "

     NUMBER        DEFINED     REFERENCES

      1200           13          10
      2000           16          21    34
      3000           18          33
      3410           25          22
      4100           28          31
      4200           32          28
      5000           34          26


OUTLINE OF "EVENT: PROCESS-DESCRIPTIONS"    04/28/06   15:57:31         PAGE   8
   IN SOURCE MODULE: CHECKER


1. <Entry>: Perform initialization.                                  *

      1.1 <Entry>: Zero IERROR; validate N.  If invalid, set IERROR=2 & rtrn

      1.2 <N valid>: Zero NMRET.

      1.3 <Argument processing completed>: Init I & J row-column counters.

2. <(0<=I<=N) & (0<=J<=N)>: Advance one square & set it to RED; drop*

3. <Counter now at I,J>: Test corners of all squares; skip ->#5 if OK

      3.1 <Entry>: Init OK=True for current testing of squares upto (I,J).

      3.2 <OK=True>: Max No. squares is one less min of (I,J); save in MINNY

      3.3 <MINNY defined>: If zero, goback ->#2 to advance one more square.

      3.4 <MINNY squares within (I,J)>: Check on colors of all their corners

      3.5 <OK will be False if corners of any square same>: But if OK, ->#5.

4. <Corners of some square same color>: Interchange & ->#3 to re-test

      4.1 <Entry>: If current (I,J) square not RED, retreat until it is.

      4.2 <Current (I,J) square now RED>: Set it BLACK & ->#3 to re-test.

5. <Not same color>: If any squares left, goback ->#2 to advance.   *

6. <All squares validly filled>: So return.                          *
```

A discussion of Dr. Anderson's algorithm in the CHECKER subroutine

Since Figure 1.a just contains a header section and the declarations for the subroutine, we can accordingly skip right to Figure 1.b to view the code logic.

Apart from the validation of `N`, and the initialization of `NMRET` and of `I` and `J` in Block #1, and the `RETURN` in Block #6, we can see that the main activity occurs within Block #s 2 through 5, which form a computational loop, in which

Block #2: The (`I`,`J`) indices are advanced to the next cell by calling the ADVANC subroutine, and `RED` is stored into that cell

Block #3: All possible squares within (`I`,`J`) are tested to see if the corners of any square are all the same color. If no corners of all possible squares contain the same color, then skip to Block #5; else drop to Block #4.

Block #4: If the current cell (`I`,`J`) is `RED`, then set it `BLACK` and go back to Block #3 to re-test. Else, keep retreating one cell at a time until a `RED` one is found, then change it to `BLACK` and go back to Block #3.

Block #5: Of all squares possible within (`I`,`J`), none contain corners all having the same color. Therefore, if any cells within the NxN checkerboard remain, go back to #2 to advance; else drop to Block #6 to return.

In brief, advance, test, interchange if needed — retreating if necessary until an interchange is possible; then go back to advance again, until all NxN squares are properly loaded.

Dr. Anderson's algorithm is elegant, yet very straight-forward. It is easy to understand.

Admittedly, we haven't yet looked at *how* the `ADVANC` and `RETRET` subroutines work. (Now you can skip over to Figure 2 on page 29 to view these modules, if you want.) But first, we are instead going to postpone that for a moment to look at the various *events* preceding each of the Functional Blocks in Figure 1.b, in order to determine if they are all *unique*.

Are all the *events* preceding each block *unique?*

From the cross reference of the statement numbers in Figure 1.c, we see that only two Functional Blocks are accessed by more than a single entry — Block #s 2 and 3, which we show in Table 2 below.

Table 2: Events Preceding Multiply-Accessed Blocks in the CHECKER subroutine

Block #	**Approached from**	<**Event**>
Block #2	Drop down from Block #1 above	`I=0` and `J=0`
	Branch back from Block #3.3	`I=1` or `J=1`
	Branch back from Block #5	`I<N` or `J<N`
Block #3	Drop down from Block #2 above	Current square at (`I`,`J`) and loaded
	Branch back from Block #4.2	Current square at (`I`,`J`) and loaded

Let's look at the entry to Block #3 first, in which we note just two points of access — a drop down from Block #2 above it, and a backward branch to it from Block #4.2. In both cases,

Figure 2: Listing of the ADVANC and RETRET subroutines.

```
     Directory: M:\GOTOs_OK                                        Input file: advanc.for
     Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)           RENUMF 2.3.0, Oct. 25, 2004
     LISTING OF SOURCE DECK "ADVANC                    "           04/28/06  18:02:37  PAGE   1

 1         SUBROUTINE ADVANC (I, J, N)                                    ADVA0010
     C                                                                    ADVA0020
     C                                                                    ADVA0030
     C*********************************************************************ADVA0040
     C*****Declarations.                                                  *ADVA0050
     C*********************************************************************ADVA0060
 2         IMPLICIT NONE                                                  ADVA0070
 3         INTEGER I, J, N                                                ADVA0080
     C                                                                    ADVA0090
     C                                                                    ADVA0100
     C*********************************************************************ADVA0110
     C(01) <Entry>: If both I & J are zero, set each to 1 and ->#3 to Return*ADVA0120
     C*********************************************************************ADVA0130
 4         IF ((I+J) .NE. 0) GO TO 2000                                   ADVA0140
 5         J = 1                                                          ADVA0150
 6         I = 1                                                          ADVA0160
 7         GO TO 3000                                                     ADVA0170
     C                                                                    ADVA0180
     C                                                                    ADVA0190
     C*********************************************************************ADVA0200
     C(02) <(I+J) <> 0>: Increment I, & J if requird, to advance to next cellADVA0210
     C*********************************************************************ADVA0220
 8    2000 I = I + 1                                                      ADVA0230
 9         IF (I .LE. N) GO TO 3000                                       ADVA0240
10         I = 1                                                          ADVA0250
11         J = J + 1                                                      ADVA0260
     C                                                                    ADVA0270
     C                                                                    ADVA0280
     C*********************************************************************ADVA0290
     C(03) <(I,J) updated to next cell>: Return.                         *ADVA0300
     C*********************************************************************ADVA0310
12    3000 RETURN                                                        ADVA0320
13         END                                                            ADVA0330


     Directory: M:\GOTOs_OK                                        Input file: retret.for
     Licensed (RN1006) for Ronald C. Wackwitz (COMP-AID)           RENUMF 2.3.0, Oct. 25, 2004
     LISTING OF SOURCE DECK "RETRET                    "           04/28/06  18:35:11  PAGE   1

 1         SUBROUTINE RETRET (I, J, N)                                    RETR0010
     C                                                                    RETR0020
     C                                                                    RETR0030
     C*********************************************************************RETR0040
     C*****Declarations.                                                  *RETR0050
     C*********************************************************************RETR0060
 2         IMPLICIT NONE                                                  RETR0070
 3         INTEGER I, J, N                                                RETR0080
     C                                                                    RETR0090
     C                                                                    RETR0100
     C*********************************************************************RETR0110
     C(01) <Entry>: Decriment I.  If >0, skip to #3 to return.           *RETR0120
     C*********************************************************************RETR0130
 4         I = I - 1                                                      RETR0140
 5         IF (I .GT. 0) GO TO 3000                                       RETR0150
     C                                                                    RETR0160
     C                                                                    RETR0170
     C*********************************************************************RETR0180
     C(02) <I=0>: Set I=N & decriment J.  If J>0, ->#3; else terminate.   *RETR0190
     C*********************************************************************RETR0200
 6         I = N                                                          RETR0210
 7         J = J - 1                                                      RETR0220
 8         IF (J .GT. 0) GO TO 3000                                       RETR0230
 9         WRITE (*,2010) N                                               RETR0240
10    2010 FORMAT (1H0 I5, ' is blocked')                                 RETR0250
11         STOP                                                           RETR0260
     C                                                                    RETR0270
     C                                                                    RETR0280
     C*********************************************************************RETR0290
     C(03) <(I,J) retreated to prior cell>: Print its value; then return.*RETR0300
     C*********************************************************************RETR0310
     C      WRITE (*,9990) I, J                                           RETR0320
     C 9990 FORMAT (1H  'Retreated back to I=',I2, ' and J=', I2)         RETR0330
12    3000 RETURN                                                        RETR0340
13         END                                                            RETR0350
```

the *event* for entry to Block #3 is the *same* — the current square is at (`I,J`), and it has had a color value freshly loaded into it. When all entry assertions to a block are identical, then we term that a very strong entry assertion.

However, we shall see that the case for the entries to Block #2 results in a much weaker overall entry assertion, due to its need to accommodate each of the three slightly different access points. In such a case, all that we can do is to arrive at a particularly general entry assertion that accommodates each of the cases — `(0<=I<=N)` and `(0<=J<=N)`. The more general we must make the composite assertion, the weaker it becomes.

For example, please note that Block #2 is accessed for `I=0` and `J=0` only for the initial drop-down from Block #1. (These values of (I,J)=0, by the way, are used to initialize the `ADVANC` subroutine, as we see from Figure 2.) Thereafter, in all subsequent accesses to Block #3, the more specific assertion of `(1<=I<=N)` and `(1<=J<=N)` applies.

The termination of the CHECKERBRD program in the `RETRET` subroutine

We really should have modified the `RETRET` subroutine, such that the termination in its Block #2 is instead passed through to an output argument, `IERROR`, which the `CHECKER` subroutine could then pick up in its call to `RETRET` in Block #4.1. That would permit the `CHECKER` subroutine to pass that through to the main CHECKERBRD program, which could then make decisions accordingly.

Having noted that, we for now bypass this issue.

The execution of the CHECKERBRD program — in 1977

Originally, back in 1977, we compiled the CHECKERBRD program using IBM FORTRAN G, and ran it on an IBM 370/155 mainframe. The results are shown in Table 3 below with respect to board size and the number of retreats required.

Table 3: Number of Retreats required for Board Sizes up to 10, on IBM 370/155

Board Size	Number Retreats Required
2x2	0
3x3	0
4x4	3
5x5	3
6x6	29
7x7	44
8x8	90
9x9	122
10x10	7,275,890

We did not run it for the 11x11 board size.

The execution of the CHECKERBRD program — in 2006

This time, we compiled the CHECKERBRD program using the 32-bit MS FORTRAN PowerStation (Version 4.0) compiler, and ran it on a 1.83 GHz Bantam PC,[16] with one GB of memory. This resulted in an unoptimizd CHECKERBRD program. Then we recompiled the program using time-optimization, with the "`/Ox`" switch. The results are shown in Table 4 below, with one distinction: the times shown for board sizes of 9 or less are the *total* times for 10,000 loops, while those times for board sizes of 10 or greater are the *actual* times, in seconds.

Table 4: Number of Retreats and timings on Bantam PC

Board Size	**Number Retreats Required**	Time, secs.	Optimized time, secs.
2x2	0	0.00	0.00
3x3	0	0.00	0.00
4x4	3	0.02	0.00
5x5	3	0.03	0.02
6x6	29	0.11	0.07
7x7	44	0.19	0.08
8x8	90	0.35	0.14
9x9	122	0.50	0.20
10x10	7,275,890	2.14	0.91
11x11	211,734,683	64.50	26.30

[16] Bantam Electronics (aka Tinkertronics), (512) 719-3560, is at 2600 McHale Court, Suite 100, Austin, TX 78758.

Acknowledgements

Dr. Peter G. Anderson

The authors wish to thank Dr. Peter G. Anderson, Professor Emeritus in the Computer Science Department of the Rochester Institute of Technology, both for the excellent and thought-provoking algorithm within the CHECKERBRD program which he provided us, as presented in his manual, *Structured Programming: Style Manual for FORTRAN Programmers*, and for his gracious and helpful attitude — both now and back in 1977. Yes, back then we enjoyed discussing with him the philosophy of structuring FORTRAN code.

Dr. William L. Kleb

The authors wish to additionally thank Dr. William (aka "Bil") L. Kleb, of the NASA Langley Research Center, for his explanation — with respect to his work at NASA — of the importance of choosing the appropriate compiler for the particular FORTRAN code used. As we have seen, even the optimization switch can result in significant increases in execution speed.

References

[1] Barry W. Boehm, "Software and its Impact: A Quantitative Assessment", *Datamation* **19**(5), (May 1973), 48–59.

[2] Charles P. Lecht, *The Waves of Change* [a pre-release review by *Computerworld* **11**(16), (April 18, 1977), 9 (Figure I- 6)].

[3] Malcolm M. Jones,"Software Notes", *Datamation* **19**(5), (May 1973), 196.

[4] Charles P. Lecht, *The waves of change* [a pre-release review by *Computerworld* **11**(18), (May 02, 1977), 14 and 15 (Figure II-4)].

[5] Martha Nyvall Jones, "HIPO for Developing Specifications", *Datamation* **22**(3), (March 1976), 112, 114, 121, 125.

[6] Gene R. Katkus, "Applying Structured Programing to Command, Control, and Communication Software Development", *Computer*, (June 1975), 43-47.

[7] Glenford J. Myers, *ReliabLe Software Through Composite Design* (New York: Manson/Charter Publishers, Inc., 1975), 117–119.

[8] L.L. Constantine, G.J. Myers, and W.P. Stevens, "Structured Design", *IBM Systems Journal* **13**(2), (May 1974), 115–139.

[9] Glenford J. Myers, *op. cit.*

[10] F.T. Baker, "Chief Programmer Team Management of Production Programming", *IBM Systems Journal* **11**,(1), (1972), 56–73.

[11] F. Terry Baker and Harlan D. Mills, "Chief Programmer Teams", *Datamation* **19**(12), (December 1973), 58–61.

[12] E.W. Dijkstra, "**Go to** statement considered harmful", *Comm. ACM* **11**(3), (March 1968), 147–148, 538, 541. [The second set of pages 147-148 is applicable.] '

[13] Donald E. Knuth, "Structured Programing with **go to** Statements", *Computing Surveys* **6**(4), (December 1974), 261–301.

[14] G.M. Weinberg, *The Psychology of Computer Programming* (New York: Van Nostrand Reinhold, 1971)

[15] E.W. Dijkstra, "The Humble Programmer", *Comm. ACM* **15**(10), (October 1972), 859–866.

[16] William Delaney (edited by Jack Stone), "Software Must Satisfy Client — Not Gratify Programmer's Ego", *Computerworld* (January 31, 1977), 18.

[17] G.J. Myers, "Composite design facilities of six programing languages", *IBM Systems Journal*, 3 (1976), 212–224.

[18] E.W. Dijkstra, in *Software Engineering - Concepts and Techniques*, Peter Naur, Brian Randall, J.N. Buxton [Eds.] (New York: Manson/Charter Publishers Inc., 1976), 30–31.

[19] Donald E. Knuth, *op. cit.*, 264–266.

[20] Donald E. Knuth, *op. cit.*, 265.

[21] Donald E. Knuth, *op. cit.*, 291.

[22] E.W. Dijkstra, "Structured Programing", in *Software Engineering — Concepts and Techniques*, Peter Naur, Brian Randall, J.N. Buxton [Eds.] (New York: Manson/Charter Publishers, Inc., 1976), 222–226.

[23] Donald E. Knuth, *op. cit.*, 275.

[24] Donald E. Knuth, *op. cit.*, 289–290.

[25] Donald E. Knuth, *op. cit.*, 294.

[26] Donald E. Knuth, *op. cit.*, 262–263.

[27] E.W. Dijkstra, personal communication to Donald E. Knuth, January 03, 1973.

[28] Richard J. Weiland, "Even PL/I Carries No 'Good Program' Guarantee", *Computerworld* (June 28, 1976), 25.

[29] Kenneth T. Orr, "Structured System Design More Vital Than Language", *Computerworld* (June 28, 1976), 25.

[30] Donald E. Knuth, *op. cit.*, 263.

[31] Donald E. Knuth, *op. cit.*, 268.

[32] Donald E. Knuth, *op. cit.*, 286.

[33] E.W. Dijkstra, personal communication to Donald E. Knuth, January 30, 1974.

[34] Donald E. Knuth, *op. cit.*, 282–283.

[35] Donald E. Knuth, *op. cit.*, 274–275.

[36] Robert E. Horn, "Information Mapping: New Tool to Overcome the Paper Mountain", *Educational Technology* (May, 1974), 5–8; "InformAtion Mapping", *Datamation* **21**(1), (January 1975), 85–88.

[37] David Frost, "Psychology and Program Design", *Datamation* **21**(5), (May 1975), 137–138.

[38] Knut Bulow, "Programing in Book Format", *Datamation* **20**(10), (October 1974), 85–86.

[39] Donald E. Knuth, *op. cit.*, 272.

[40] Harlan D. Mills, "Top-down Programming in Large Systems", in *Debugging Techniques in Large Systems*, R. Rustin [Ed.] (Englewood Cliffs, N.J.: Prentice-Hall, 1971), 41–55.

[41] Ronald C. Wackwitz and Jay Falck, *Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)*. Technical Report, The COMP-AID Company, 513 Old Bear Creek Road; New Braunfels, TX 78132, February 1980, 87 pages.

[42] Peter Gordon Anderson, "Structured Programming: Style Manual for FORTRAN Programmers", (Newark: New Jersey Institute of Technology, September 1976), 10–12.

[43] Andrew Scriven, "Scientific Programming in Fortran 90 with the IBM xlf90 Compiler", *Fortran Journal* **6**(2), (March/April 1994), 6.

[44] Niklaus Wirth, "Program development by stepwise refinement", *Comm. ACM* **14**(4), (April 1971), 221–227.

[45] Scott W. Ambler, "How Agile Are You?", *Software Development* **13**(12), (December 2005), 47–49.

[46] William L. Kleb *et al*, "Collaborative Software Development in Support of Fast Adaptive AeroSpace Tools (FAAST)", http://library-dspace.larc.nasa.gov/dspace/jsp/bitstream/2002/12651/1/NASA-aiaa-2003-3978.pdf (November 2003), 6–12.

[47] William L. Kleb *et al*, *op. cit.*, 17–19.

[48] Andrew Scriven, *op. cit.*, 16.

[49] William L. Kleb, personal communication to Ronald C. Wackwitz, April 7, 2006.