FOR-WORD

FORTRAN DEVELOPMENT NEWSLETTER

VOLUME 1
No. 1 - Feb. 1975
thru
No. 6 - Feb. 1976

Loren P. Meissner

LAWRENCE BERKELEY LABORATORY
BERKELEY, CALIFORNIA 94720

Prepared for the U. S. Energy Research and
Development Administration under Contract W-7405-ENG-48

DISCLAIMER

"FOR-WORD" Fortran Development Newsletter

Volume 1

by

Loren P. Meissner

Lawrence Berkeley Laboratory
Berkeley, California

Introduction

The need for communication among those interested in the development of the Fortran programming language was recognized as early as 1974, by such persons as Donald J. Reifer and Guy J. De Balbine. Inspired by these individuals and others, Loren P. Meissner decided in February, 1975, to produce a newsletter for information concerning developments in this language. The first three issues were only one or two pages long, but as more interested persons began contributing material, and as liaison with the ANSI X3J3 Fortran Standards Committee increased, the amount of material considered to be of interest to the readers of the newsletter increased as well.

This report is a compilation and reprint of the first six issues of the newsletter, which were distributed between February 1975 and January 1976.

= F O R - W O R D = >

Fortran Development Newsletter

Volume 1

(Reprint)

CONTENTS

Note: Two other documents were distributed during 1975 to addressees on the "FOR-WORD" mailing list. These are:

"On Extending Fortran Control Structures to Facilitate Structured Programming," by L. P. Meissner; see SIGPLAN Notices Vol. 10, No. 9, September 1975, page 19

"Proposed Control Structures for Extended Fortran," by L. P. Meissner; see SIGPLAN Notices Vol. 11, No. 1, January 1976, page 16

Copies of these papers are available from L. P. Meissner.

No. 1 1 Feb 75

NEWSLETTER of
THE WEST COAST WORKING GROUP ON FORTRAN LANGUAGE DEVELOPMENT

1. History.

Beginning at the 1974 Lake Arrowhead Workshop on Structured Programming, and building up steam at the Workshop on Fortran Preprocessors for Numerical Software, has been the idea of establishing some sort of intercommunication facility for people interested in trends in Fortran language development, beyond the level now under consideration by the Standards group (ANSI - X3J3). Of primary concern at the present time is the question of Fortran language extensions to accomodate structured programming.

Recently Donald J Reifer of Aerospace Corporation has proposed the formation of a small working group on the West Coast, to study the question of programming in Fortran or in an extended Fortran dialect, in accordance with the "principles of Structured Programming."

One notable trend is the proliferation of preprocessors, which accept various source language dialects more or less like Fortran, and produce "standard Fortran" code. The preprocessor input languages are designed, in many cases, to accept a language that includes such keywords as "WHILE" or "ELSE," which are supposed to be the hallmark of a structured programming language. Reifer and others are asking whether it is possible to discern one or more areas of concensus among these dialects.

Dr. Guy de Balbine has taken the initiative in compiling a mailing list of persons who have declared to him their interest in the future of Fortran as it relates to Structured Programming. I have expanded this list somewhat, and am taking the further initiative of mailing out what may be, for a while at least, a monthly newsletter discussing the activities of this working group, if it continues to exist. (Initial mailing: 64 copies.)

2. Announcements.

Wednesday, February 12, at 6 p m, at the Holiday Inn - Westwood (near UCLA) there will be a meeting of a small group of persons to discuss the possibilities of organizing such a group as has been proposed by Don Reifer. Anyone interested in attending should call Don at 213-648-6260. This is the evening before the Computer Science and Statistics Symposium at UCLA.

Thursday, February 20, at 7 p m, at the Statler Hilton, Washington DC there will be a Special Interest Session on Structured Fortran. The chairman will be L. P. Meissner. This is in connection with the SIGCSE Symposium. Also on the next day, Professor Gibbs of William and Mary will be the chairman of a formal session on Structured Fortran. This session will include papers by W. R. Bezanson of Carleton University, J. L. Wagener of SUNY Brockport, and L. P. Meissner.

Wednesday, Februray 26, in the evening, at the Jack Tar Hotel, San Francisco, there will be the second meeting of the West Coast Working Group on Structured Fortran. It is expected that several interested persons from outside California will be able to attend this session, which is at the same place as "COMPCON," the Spring meeting of the IEEE Computer Society.

3. For Further Information: Dr. Loren P. Meissner (50-B 3239), Lawrence Berkeley Laboratory, Berkeley, CA 94720. 415-843-2740, ext. 6361. Or call Don Reifer at the number mentioned above.

"F O R - W O R D"

Fortran Development Newsletter
No. 2 -- 24 Mar. 1975

Mailed by -- Loren P. Meissner (50-B 3239)
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

1. The Structured Fortran Working Group:

Met in Los Angeles on 12 Feb., and agreed to publish a language feature catalog within six months, covering existing Fortran preprocessors. To this end, Don Reifer is soliciting information concerning additional existing structured Fortran implementations. Later, the group expects to recommend preferred forms for the representation of various extended control constructs. The group met again in San Francisco on 26 Feb., and discussed a skeleton list of language features to be used as a basis for comparing structured Fortran dialects.

The next meeting of this group will be at the International Hotel, Los Angeles, during the time of the International Conference on Reliable Software (April 21 - 23). Besides a working meeting for those involved in the survey, an open progress reporting session will probably be held. Time and place of meetings will be posted in the conference registration area.

Don Reifer has submitted two papers for publication by several of the ACM Special Interest Groups. One of these, entitled "The Structured Fortran Dilemma," outlines the problem that led to the formation of the Structured Fortran Working Group. The other paper, "The Structured Fortran Working Group," gives further information concerning the preprocessor survey being undertaken by the group. Watch your SIG news bulletins for these papers.

Continue to report any structured Fortran dialects that you may hear about, either to Loren P. Meissner (address above) or to Donald J. Reifer, MS 100-2034, The Aerospace Corporation, PO Box 92957, Los Angeles, CA 90009.

To simplify communications, all of the active members of the working group have been recruited from the West Coast. They are Terry Beyer (Eugene OR), Don Buchwald (Los Angeles), Loren Carpenter (Renton WA), Martin Cohen (Marina del Rey CA), A. James Cook (Stanford CA), Guy de Balbine (Pasadena CA), Ellis Horowitz (Los Angeles), Charles L. Lawson (Pasadena, CA), Edward F. Miller, Jr. (Santa Barbara, CA), Jock Rader (Culver City, CA), Donald J. Reifer (Los Angeles), and Loren P. Meissner (Berkeley, CA).

2. Special Interest Session on Structured Fortran at SIGCSE Symposium, Washington, DC (20 Feb.)

Loren P. Meissner was chairman of this session, which was devoted to discussion of pedagogical and educational implications of structured Fortran extensions. Meissner presented a preliminary survey of control structures found in 14 existing dialects, and discussed the need for multiple level exits and the implementation of the Dahl and Zahn constructs mentioned by Knuth (Comp. Surveys, Dec. 1974). Other points covered during the meeting included the need to introduce control structure discipline early in the student's career, and the disadvantages of preprocessors for student use. James Vandendorpe of Illinois Institute of Technology, and Selden L. Stewart of the National Bureau of Standards, described preprocessors that had not previously been published.

3. Cooperation with the ANSI X3J3 Fortran Standards Committee

The committee is nearing completion of a revised standard, and expects to present it publicly early in 1976. Frank Engel, Jr., who is the chairman of this committee, met with three members of the West Coast group recently. Frank recalled that after publication of the original standard in 1966, there was very little interest in further development of the language. The general feeling seemed to be that Fortran had now been standardized and was ready to settle down, while PL/I was rising in popularity and might take over before any further need for Fortran development could occur. The situation seems to be quite different this time. The committee has a list of proposed additions to Fortran that might be worthy but could not be considered in time for the current revision. Also, there are some active groups (including the West Coast Structured Fortran group, and others) who are interested in the further development of Fortran in specific directions.

In view of the pending completion of this phase of its work, and of the apparent interest in Fortran within the user community, the standards committee might like to present its new proposals at an open meeting some time early next year, and would welcome support from the user community in organizing such a meeting. Meissner suggested the possibility of a West Coast presentation next February in connection with the Computer Science Conference, followed by an East Coast presentation in connection with the National Computer Conference at New York in June. Further exploration of the first possibility has already been undertaken, and planning has begun for a Fortran forum at Anaheim

during the week of 9-13 Feb. 1976. The plan is to have a one day meeting on each coast, and to include in each some presentations by members of the X3J3 committee as well as some invited papers by others working on Fortran language developments in specific areas of interest. Sponsorship by the local ACM chapters and possibly by SIGPLAN is being explored. Volunteers are needed to help with the details of both meetings.

East Coast readers of this Newsletter please note: plans for the June 1976 Fortran Forum East need to get under way soon. Let's discuss this next month in Los Angeles (at the conference 21-23 Apr.) or write to me or to Frank Engel Jr., 179 Lewis Rd., Belmont, MA 02178 (617 484-5911). Nominations for meeting chairman are in order.

Continuing efforts. Besides the need to publicize and implement the revised ANSI Fortran standard, it was noted at the recent meeting with Frank Engel that efforts are needed in two other areas on a continuing basis. First of all, the ANSI X3J3 committee will be recruiting new members to carry the torch of further standards revision. The next revision will be due in five years, and judging from past experience it will take at least six years to produce it. There seems to be some possibility of organizing a subcommittee even before publication of the current revision, to consider the left-over proposals and begin working on control structures and other features for next time around.

There remains the question of whether a separate Fortran language development effort, which would concern itself with a look further into the future than the standards committee, is needed. As a first step in this direction, I am exploring with SIGPLAN the possibility of forming a Fortran subcommittee within that organization. SIGPLAN seems to me to be the most logical home for such an effort. Any comments?

Besides the West Coast Structured Fortran group, there is now an "Industrial Fortran Group" concerned with process control applications; also a "Working Party on Fortran Extensions" in the British Computer Society, and a Data Base Management Group in Codasyl with some interest in Fortran development. Other groups that might be interested include the computer user groups (Share, etc.), BEMA and ECMA (European) manufacturers' groups, and of course the software groups working for the computer manufacturers themselves. The National Bureau of Standards has some responsibility for Fortran standards within the U.S. government. In fact, every major computer user has considerable stake in the future of Fortran. It should be possible to tap some of this interest, in order to get some communication going on the subject of future directions for development of the Fortran language.

Financial support seems to be a more difficult problem. Although many organizations have a lot to gain from the orderly development of the Fortran language, it is hard to get this fact converted into support for standardization and development efforts. The best that can be managed seems to be the time and travel expenses of a few volunteers who work on committees. While these efforts are far better than nothing, it may be that more energy is required for the task than can be expected on a volunteer basis. Obviously, domination of language development efforts by the manufacturers would be undesirable. Support from major users might work out better, if it could be arranged. The federal government seems willing to support in-house efforts in certain areas, such as the testing of software procurements for conformance with existing standards, but so far seems not to have taken responsibility for development of new language features. Research grants are aimed at specific projects, rather than at cooperative efforts to organize language developments. The only glimmer of professional society support appears in the ACM Long Range Planning Committee report (Comm. ACM Feb. 1975) which includes some statements of the desirability of a more active role for the organization in this area. It is not clear how much this will mean in practice, or how soon it will become effective. Meanwhile, if most of the work must continue to be done by volunteers, then more people should get involved so that the burden on any one individual can be decreased.

Structured Fortran Bibliography (I) [More next time. Contributions welcome.]

Miller, E.F. Extensions to Fortran to support structured programming. SIGPLAN Notices 8, 6 (Jun 73)

Miller, L.R. Linus, a structured language for instructional use. SIGCSE Bulletin 6, 1 (Feb 73)

Hull, T.E. Would you believe structured Fortran? SIGNUM Newsletter 8, 4 (Oct 73)

Rosin, R.F., and Hull, T.E. Correspondence on structured Fortran. SIGNUM Newsletter 9, 2 (Apr 74)

Meissner, L.P., A compatible structured extension to Fortran, SIGPLAN Notices 9, 10 (Oct 74)

Ralston, A. The future of higher level languages (in teaching). Proc. Int. Comp. Symposium 1973

Tenny, Ted Structured programming in Fortran, Datamation 20, 7 (Apr 74)

O'Neill, D.M. Sfor - A precompiler for the implementation of a Fortran-based structured language. SIGPLAN Notices 9, 12 (Dec 74)

= F O R - W O R D = >

Fortran Development Newsletter No. 3 1 May 1975

Mailed by -- Loren P. Meissner (50-B 3239)
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

- -

1. The Structured Fortran Working Group

The third meeting of this group was held at Los Angeles on April 22, 1975. Don Reifer reported that he is working with the appropriate groups, on sponsorship of a Fortran Forum at Anaheim in February 1976. This is to be a one-day presentation, overlapping with the Computer Science Conference or the SIGCSE Symposium during the week of February 9 - 13, 1976. It is to include discussion of the new ANSI Fortran standard (led by members of the X3J3 committee) as well as invited papers on subjects related to the future of Fortran. Plans will proceed as soon as the necessary approvals are obtained.

The Working Group discussed progress toward gathering information concerning existing preprocessors and other language extensions for structured programming. A committee was appointed to refine the proposed questionnaire to be sent to the developers of extended Fortran dialects.

Further discussion centered on the clarification of the objectives of the group, and on the relation of preprocessor input languages to Standard Fortran or to future extended compiler input languages.

2. Toward a Fortran Development Committee

Between sessions of the International Conference on Reliable Software in Los Angeles (Apr. 21 - 23), the idea of forming a Fortran Development Committee was discussed. Dr. Halstead, the current Chairman of SIGPLAN, stated that his group would probably be interested in a proposal for the formation of such a group as a Special Technical Committee of SIGPLAN, more or less parallel to the existing committee on APL.

The executive board of SIGPLAN will be meeting at the NCC in Anaheim, and a proposal could be brought before them at that time. This will require a statement of objectives, a set of by-laws (which could be based on those of the APL committee), a slate of officers, and the signatures of as many persons as possible who are in favor of the establishment of such a committee.

A form for gathering signatures is enclosed with this Newsletter. SIGPLAN member signatures are preferred, otherwise ACM members would be nice; but the declaration by any interested person of support for the idea of such a committee will have some weight. Please feel free to make copies of the form, or attach signatures on blank paper, or use the reverse side. Signatures that are received by Loren Meissner by 15 May will be presented at the NCC. Other signatures will be helpful as well, in possible further discussions with the SIGPLAN board. (One possibility is that some, but not all, of the necessary requirements for the establishment of the committee will be met in time for the NCC meeting.)

Unfortunately, a petition is a rather poor format for expressing the negative of the proposed opinion. If you are opposed, especially if you have definite ideas or alternative suggestions, send them to any of the proposed officers (listed below) or to a member of the SIGPLAN executive board (listed inside the cover of SIGPLAN Notices).

Proposed interim officers, to serve until an election is held under the by-laws, are

Chairman: Paul B. Schneck, New York NY
Vice Chairman: Guy J. de Balbine, Pasadena CA
Secretary-Treasurer: Michael A. Malcolm, Waterloo, Ontario
Editor: Loren P. Meissner, Berkeley CA

One possibility is that "representatives" from various groups working toward Fortran development will be recognized and given a definite status with regard to the committee. Suggestions are welcome as to means for accomplishing this, and as to various groups that should be contacted for their views on subjects of interest to the committee.

PLEASE SIGN AND RETURN YOUR PETITION RIGHT AWAY.

3. On Preprocessors. (Editorial?)

One attendee at the Structured Fortran Working Group meeting expressed the opinion that it may be just as important to emphasize portability and usability of the output produced by a preprocessor, as to attempt to standardize preprocessor input. Further discussion after the meeting has led to the following conclusion in this regard.

There are two ways to transport a "structured" Fortran program that has been developed with the aid of a preprocessor. One way is to transport the preprocessor *source* code *along with* the preprocessor itself. This requires that the preprocessor be extremely portable, so that the user can implement it on his equipment with virtually no effort and at little expense. The program can then be maintained in preprocessor source form. The other way is for the originator to use the preprocessor to *convert* the structured program to portable Fortran and transport the preprocessor *output*. To make this approach feasible, efforts would be required to make the Fortran language output from some of the preprocessors more readable, so that it could be used as a basis for code maintenance by the user. In either case, it is not clear why the preprocessor source language should resemble Fortran (rather than Pascal, for example), except to permit some gain in preprocessor efficiency when source statements are recognized as being Fortran statements.

Paul Jensen writes, "Because of the many structured Fortran dialects, that will probably continue to exist for some time, it appears that the production of very readable Fortran "object" programs by the preprocessors is of key importance."

= F O R - W O R D = >

FORTRAN DEVELOPMENT NEWSLETTER

No. 4 -- Published by
Ad hoc Committee on Fortran Development
ACM - SIGPLAN

Table of Contents

CALENDAR

25-29 Aug 75 Meeting of ANSI Fortran Standards Committee at Bell Labs, Holmdel NJ. Contact: Tom Gibson, 2G 428.

8-12 Sep 75 COMPCON Fall 75, and Conf on Software Engineering; Mayflower Hotel Washington DC. For information write COMPCON, PO Box 639, Silver Spring MD 20901

6-10 Oct 75 Reserved for possible ANSI Fortran Standards Committee mtg.

20-22 Oct 75 ACM 75 National Conference Radisson Hotel, Minneapolis MN. Chairman: Earl Joseph, PO Box 3525, St Paul MN 55165

17-18 Nov 75 Fourth Texas Conf on Computing Systems, Austin TX. Chairman: SK Basu, Computer Sci Dept, Univ of TX, Austin TX 78712

19-21 Jan 76 Symp. on Principles of Programming Lang., Atlanta GA. Prog. Chairman: SL Graham, Computer Sci Division, Univ of Calif, Berkeley CA 94720

9 Feb 76 FORTRAN FORUM WEST at Disneyland Hotel, Anaheim CA. Chairman: Donald J Reifer, Aerospace Corp. PO Box 92957, Los Angeles 90009

10-12 Feb 76 Computer Science Conf 76, Anaheim CA. Chairman: Julian Feldman, Dept of Inf & Computer Sci, Univ of Calif, Irvine CA 92664

22-24 Mar 76 Conf on Data, Abstraction, Definition, & Structure, Salt Lake City, UT.

7-10 Jun 76 National Computer Conf 76, New York.

7-10 Jun 76 Possible FORTRAN FORUM EAST at National Computer Conf 76. Tentative Chairman: Paul B Schneck, Goddard Institute, 2880 Broadway, NY 10025

NEWSLETTER ITEMS SOLICITED

Send Calendar Items, Bibliographies or Reviews, Correspondence, Comments, Thoughts on Fortran Development, etc. to the Editor:

Loren P. Meissner
50-B 3239
Lawrence Berkeley Laboratory
Berkeley CA 94720

SIGPLAN Fortran Development Committee

SIGPLAN (ACM Special Interest Group on Programming Languages) has been asked to create a Special Technical Committee on Fortran Development. A proposal for creation of such a committee was presented to the SIGPLAN Executive Committee in Anaheim in May 1975. The presentation included well over 50 petition signatures, an interim slate of officers, a proposed set of bylaws, and a statement of the purpose of the committee.

Although final action is pending, awaiting further action by ACM, the SIGPLAN Executive Committee authorized the appointment of the proposed officers as an ad hoc committee under SIGPLAN sponsorship, to take further steps toward formalization of a SIGPLAN Technical Committee on Fortran Development, and to "establish activities consonant with the existence of such a technical committee." Some financial assistance was also promised.

The petition signatures presented in May were obtained in less than two weeks as the result of a single mailing. Additional petitions have been received since May, bringing the total to about 150, of which about 100 signatures bear an ACM Membership Number.

The purposes of the committee include: To form a bridge between active implementations of Fortran extensions, and the ANSI X3J3 Fortran standards effort. To provide a forum for interchange of ideas and proposals, for groups interested in extending the Fortran language in various ways. To obtain and distribute information concerning developments in, and extensions to, the Fortran language as defined by existing standards. To take stands and make recommendations (after adequate consultation with all interested parties) concerning the desirability or undesirability of implementing certain features in certain ways, in the hope of reducing the proliferation of dialects by reacting in a timely way to new language developments.

The ad hoc committee, as currently constituted, consists of the following persons who are to become interim officers of the Technical Committee when it is formally established: Paul B. Schneck, New York NY (Chairman); Guy J. de Balbine, Pasadena CA (Vice Chairman); Michael A. Malcolm, Waterloo Ont (Secretary-Treasurer); and Loren P. Meissner, Berkeley CA (Editor). The Editor is now maintaining a mailing list, which will eventually form the basis for a Membership List of the Technical Committee. It is expected that, after a period during which activities of the group are supported by SIGPLAN, it will eventually be necessary to charge a membership fee to cover the cost of distributing the Newsletter. -LPM

The West Coast Working Group on Structured Fortran

Current activities of this group include organizing the Fortran Forum West (reported elsewhere in this Newsletter), and conducting a Preprocessor Survey.

A committee met recently to decide upon a final format for a Survey Questionnaire to be mailed to the developers of all known Structured Fortran preprocessors. (Although the committee is interested in Fortran extensions that provide structured programming features by other means than preprocessors, it was found to be nearly impossible to develop a single composite survey form of reasonable length, that would cover compiler and interpreter implementations, for instance, as well as preprocessors.)

The committee hopes to receive the results of this survey in time to tabulate them for distribution at the Fortran Forum West in February 76.

A list of the Structured Fortran implementations currently known to the group appears elsewhere in this Newsletter. Additions to this list are solicited. Please send information to Donald J. Reifer, MS 100-2034, The Aerospace Corporation, PO Box 92957, Los Angeles CA 90009.

Fortran Forum West

A one-day public session will be held on 9 February 1976, for the purpose of presenting and discussing the forthcoming Revised Fortran Standard proposed by the ANSI Fortran committee (X3J3). Portions of the meeting will also be devoted to discussion of Fortran developments in two specific areas, namely Structured Fortran and Industrial Fortran (process control).

The morning session (8 to 12) is being organized by Walt Brainerd of Pasadena CA, and will consist of presentations and discussions by members of the ANSI Fortran committee. They will report on the progress of the committee toward development of a Revised Fortran Standard. It is to be expected that a draft of the proposed revised standard will have been approved by the committee and released for public review and comment prior to the Fortran Forum. Plans are under way to make copies of the draft proposal available to all attendees at the meeting, or sooner, if possible.

The first afternoon session (1 to 3) will present Structured Fortran developments; Loren Meissner of Berkeley will be chairman. The second afternoon session (3 to 5) will be devoted to Industrial Fortran developments; Maxine Hands of San Diego CA will be the chairman of this session.

The meeting will be held at the Disneyland Hotel, Anaheim CA. The date of the Fortran Forum is 9 February 1976, which is the day preceding the ACM Computer Science Conference (10-12 Feb) to be held at the same location. The list of sponsors is expected to include the ACM and IEEE, as well as the Los Angeles ACM Chapter and several Special Interest Groups and Committees.

General Chairman is Donald J. Reifer, MS 100-2034, The Aerospace Corporation, PO Box 92957, Los Angeles CA 90009.

Fortran Standards Activity

Apparently many people are not aware of the current status of standards for the Fortran language. This is not too surprising, because very little is published on this subject. Datamation articles by Thorlin in 1972 and by Engel in 1974, and an article in SIGPLAN Notices by Engel in 1973, are perhaps the only widely accessible publications giving insight into the current work of the Fortran Standards Committee (ANSI X3J3), since the 1966 standard and its interpretations were published. One of the purposes of this Newsletter is to make more information available concerning the status of Standard Fortran.

As one symptom of the current situation, there are a lot of Fortran programmers who are not sure which features are or are not in the current (1966) Standard Fortran language. One common misconception relates to the DO loop in cases like the following:

```
      LIM = 0
      DO 10 I = 1, LIM
        A(I) = 5.0
10      CONTINUE
```

A lot of people are surprised to learn that the 1966 standard does not specify that the effect of this loop is to assign the value 5.0 to A(1). The standard merely states (Section 7.1.2.8) "The value represented by the initial parameter ... must be less than or equal to the value represented by the terminal parameter." In other words, the effect of the above loop is not defined. Of course, nearly every Fortran implementation executes the body of the loop once before making the test, so the "de facto" situation (but not the situation prescribed by the standard) is that the value 5.0 will be assigned to A(1). This situation is under active review as part of the current standards revision effort.

The average programmer can be excused for not being aware of technical details of this kind, or of just what the standard says about side effects of functions, or what "second level definition" means. But often people who should know better make statements, like the one I heard recently, that language X is better than Fortran because Fortran never checks array bounds. The same speaker had just waxed indig-

nant a few minutes earlier, because his favorite language X had been criticized on another point which related to the implementation of X rather than to the language itself. And many authors have written textbooks purporting to describe "Fortran" which actually describe only a particular non-standard dialect.

The very first step in becoming knowledgeable about Fortran standards, then, should be to read the 1966 standard. The official document is available from American National Standards Institute, 1430 Broadway, NY 10018. Ask for X3.9-1966. Unfortunately, ANSI's main source of income is from the sale of standards documents. Although each copy costs only a few dollars, this makes it expensive to distribute copies of the standard widely for educational purposes, such as by reprint in an appendix to a Fortran textbook. Drafts, which were published in ACM Communications in Oct 64, May 69, and Oct 71, should be available in most technical libraries.

The next step is to read what is available concerning the recent work of the ANSI X3J3 committee. The best single source is Engel's article in SIGPLAN Notices, Mar 73. This article tabulates 80 proposed modifications in 6 categories. There have been few changes in the position of the committee since this list was prepared. The most important are the deletion of array operations and of binary data type. A draft proposal for a revised standard is nearly ready for publication. Information concerning this document, when available, will be published in this Newsletter, in SIGPLAN Notices, and in the ACM Communications. Plans are also under way for two Fortran Forum workshops, one in the Western US in Feb 76 and one in the East in Jun 76, to provide public feedback to the committee concerning standards revision proposals.

The very best way to keep up to date on the committee's activities is to attend committee meetings as an observer. If you are truly serious about your interest in Fortran, you can even apply for membership. However, this entails a fair amount of travel, at an expense of a few thousand dollars a year per member. Actually this expense is probably well justified at most installations, considering the large amount of their investment in Fortran and the possible effect of changes in the Standard; but it is a point on which most managers are hard to convince: travel expenses are generally rather closely watched. But the meetings are held at various places around the country, and you should be able to attend some of them. This Newsletter will try to include ANSI X3J3 committee meetings on its Calendar. The meetings are open to the public, but facilities are usually somewhat limited, so advance notice of your intentions will be appreciated.

If you can't attend the meetings of the ANSI X3J3 Fortran standards committee, you can ask to have your name added to the committee's mailing list. Write to the vice chairman of the committee, Martin N. Greenfield, MS 824A, Honeywell Info. Sys., 300 Concord Road, Billerica MA 01821, asking to be included on the distribution list and stating reasons for your interest in the committee's work. Material distributed by the committee includes complete minutes of all meetings, and periodically the latest approved drafts of the working documents (potential draft standards). In particular, the distribution includes those documents submitted to committee members for formal vote.

Secondary sources of information concerning the workings of the committee include SIGPLAN Notices, also the newsletters of ACM special interest groups such as SIGNUM and SIGCSE; Datamation, Software Engineering (IEEE Computer Society), Software Practice and Experience, and Communications of ACM. This "FOR-WORD" Newsletter aims to provide a somewhat more concentrated source for Fortran standards development information than any that has been available.

Standard Industrial Fortran for Process Control

Largely as a result of the work of the International Purdue Workshop on Industrial Computer Systems, whose chairman is Theodore J. Williams, Director of Purdue Laboratory for Applied Industrial Control (at Purdue University), the Instrument Society of America has adopted a standard for executive functions and process input-output, for industrial users of Fortran. The standard is available as ISA-S61.1-1972, from Instrument Society of America, 400 Stanwix Street, Pittsburgh PA 15222. It defines names and calling sequences for a number of subroutines useful in Fortran programs for industrial control. Active in the development of this standard have been Mrs. Maxine Hands of San Diego CA, chairman of the Fortran committee of the International Purdue Workshop, and Mr. Matthew Gordon-Clark of Philadelphia PA, chairman of the Real-Time Fortran Committee of the Instrument Society of America.

SIGPLAN Notices to publish Draft of Proposed Revised Fortran Standard

The Fortran Standards Committee (ANSI X3J3) and ACM have tentatively agreed that the full text of the forthcoming Draft Proposed Fortran Standard (Revised) will be published in SIGPLAN Notices, perhaps in January 1976. Members of the committee plan to prepare summary articles highlighting features of the revision, for publication in Communications of the ACM and elsewhere.

Workshop on Fortran Preprocessors for Numerical Software

Jet Propulsion Laboratory; November 7-8 1974
Reported by C.L. Lawson

(Reprinted with permission from SIGNUM Newsletter, Vol. 10, No. 1, Jan 1975)

This workshop, cosponsored by SIGNUM and JPL, was attended by 110 persons of whom sixty were from outside the Los Angeles area.

The program featured thirty-one speakers plus a final discussion period moderated by Tony Ralston. The four half-day sessions were chaired with sensitivity and aplomb by John Rice, Ellis Horowitz, Dick Lau, and Bob Mercer.

My general impression, and this was also expressed by numerous attendees, was that the workshop was timely and very worthwhile for those working on enhancements to the Fortran language via preprocessors or other language developments in which Fortran is used as a universal machine language.

The concept of adding control statements for structured programming in Fortran has now been explored by many persons. Some Structured Fortrans have reached a fairly mature level at which the implementors are supporting a user community doing real applications. Other Structured Fortrans are more exploratory, being used as test vehicles for various control structures. Activities of both of these types should accelerate the processes by which a wider community of programmers and managers gain experience with the working realities of structured programming. I hope that in due time some of these enhancements will evolve as the preferred ones and will reach a status of being de facto standards, followed eventually by ANSI standardization.

A concept which is at the heart of the portability problem is the ability to indicate changes which must be made in a code for transportability of the code to different computer systems and the availability of a processor which can mechanically make these changes. Five of the papers, Aird, Boyle, Ford, Krogh, and Walsh, reported on systems which attack this problem. I think systems of this type provide the best prospect for transportability of sophisticated efficient mathematical software in the near future.

A survey of portability techniques with suggested guidelines for longer range development of a portability support system was given by Malcolm.

A common tie connecting all participants in the workshop was the choice of Fortran as a base language for projects in which portability was a key goal. This is a testimony to the value of the 1966 ANSI standard for Fortran. At the same time, most of the work reported at the workshop could be interpreted as defining changes which the speakers would like to see made in Fortran. Against this background Engle's talk on the ANSI Fortran standardization process was received with great interest and stimulated an extended discussion period.

Engle made the point that whereas COBOL has a language development committee, (Codasyl-PLC), as well as an ANSI standardization committee, there is no language development committee for Fortran. The ANSI Fortran committee X3J3 does not feel that it is chartered to actively explore extensions to the language. Aside from the policy question, X3J3 simply has no resources for language development.

Committee X3J3 is voluntary operation whose members currently hold week-long meetings every two months. They find that it is difficult with that level of effort just to complete the writing of a revised standard based primarily on language features which are already supported by many actual Fortran compilers. At present they anticipate publication of a proposed new standard about September, 1975, with final announcement of the new standard about six months later.

Other problem areas addressed at the workshop included dynamic storage allocation, program validation aids, and provision for different data types such as vectors, controlled precision numbers, error-indicating numbers, and rational numbers.

A proceedings containing a one-page summary by each author is available on request from C.L. Lawson, MS125-109A, Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, Calif. 91103. Following is a list of the presented papers plus three papers presented by title only.

Dynamic Storage Allocation

John E. Ekelund, Jet Propulsion Laboratory, "Dynamic Storage Allocation in Spacecraft Navigation Software"

Paul S. Jensen, Lockheed Palo Alto Research Laboratory, "The Impact of a Storage Manager on Program Design"

Henry Kleine, Jet Propulsion Laboratory, "Implementing Dynamic Storage Capability Without New Language Constructs"

Different Data Types

Alan L. Craig, Naval Weapons Center, "Variable Precision Computation with Floating-Point Numbers"

Fred Crary, University of Wisconsin Madison,

Workshop on Fortran Preprocessors for Numerical Software (cont.)

"AUGMENT - A General Fortran Extension"

Kirby W. Fong and Thomas L. Jordan, University of California Los Alamos, "Vectors: Why and How Should this Data Structure be Introduced?"

Don J. Orser, National Bureau of Standards, "FORPAK: A Portable General Purpose Fortran Preprocessor"

James R. Pinkert, University of Tennessee, "The Use of SAC-1 in Numerical Computations"

Debugging and Validation Aids

Herbert S. Bright, Computation Planning, Inc. "A Method of Testing Programs for Data Sensitivity"

Leon Stucki, McDonnell-Douglas Corporation, "Self-Metric Software for Debugging and Validation"

Transportability Systems

Thomas J. Aird, International Mathematical and Statistical Libraries, Inc., "Portability of Mathematical Software Coded in an ANSI Based Fortran"

James M. Boyle and Kenneth W. Dritz, Argonne National Laboratory, "A Syntax-Directed Fortran Preprocessor and Formatting System"

Brian Ford and Steven Hague, Oxford University Computing Laboratory, "NAG Numerical Software Maintenance - The Predictor Corrector Approach"

Fred T. Krogh, Jet Propulsion Laboratory, "A Preprocessor To Specialize Fortran Code"

Michael Malcolm and Lawrence Rogers, University of Waterloo, "Designing a Portable Preprocessor"

Robert T. Walsh and Nancy Ruiz, Sandia Laboratorise, "Version Selection and Storage Minimization"

Structured Fortran

Terry Beyer, University of Oregon, "FLECS: Structured Programming in Fortran"

John Flynn, Jet Propulsion Laboratory, "SFTRAN - Structured Fortran"

Brian W. Kernighan, Bell Laboratories, "RATFOR - A Rational Fortran"

Loren P. Meissner, University of California Berkeley, "A Compatible Structured IF for Fortran"

J.L. Wagener, SUNY College at Brockport, "IF - FI DO - OD Structured Fortran"

Conversion of Fortran Code to Structured Fortran

Guy de Balbine, Caine, Farber & Gordon, Inc., "Automatic Restructuring of Fortran Programs"

General Fortran-based Language Extensions (Structured programming, macros, conditional compilation, precision control, and other features)

Anders Beckman and Tom Smedsaas, Uppsala University, Sweden, "Experiments with Fortran Preprocessors"

M.A. Brebner and H.S.B.Hoover, University of Calgary, "A Base Language (Fortran?) and Macro Facilities for Mathematical Software - First Thoughts"

A. James Cook, Stanford Linear Accelerator Center, "MORTRAN2"

Robert C. Gammill, Rand Corporation, "GPMX: General Purpose Macrogenerator eXtended Used as a Fortran Preprocessor"

John Gary, University of Colorado, "A Macro Preprocessor for a Fortran Dialect"

C.W. Gear and L. Lopez, University of Illinois, "Software for Scientific Packages"

T.E. Hull, University of Toronto, "Structured Programming, Fortran and Preprocessors"

David J. Kennison, National Center for Atmospheric Research, "FRED - A Fortran Preprocessor"

Jack Perrine, Jet Propulsion Laboratory, "The ATHENA Fortran Compiler"

Arnold A. Schwartz, Jet Propulsion Laboratory, "IPLFOR - An Extensible Fortran Preprocessor"

Lenny Shustek and C.T. Zahn, Jr., Stanford Linear Accelerator Center, "Records and References in MORTRAN2"

ANSI Fortran Standardization

Frank Engel, Individual Consultant, "Development of Fortran Standards"

= =

A Quote: (R. F. Rosin, SIGCSE Bulletin, Mar-Apr 1971):

"Fortran is dead."

List of Structured Fortran Processors

For well over a year, Don Reifer has been assembling a list of processors for Structured Fortran extensions. Here is his current list, along with the name and address of the person who can provide further information. All of these processors will be included in the planned Survey, the results of which will be published by the West Coast Working Group on Structured Fortran.

IFTRAN-1 W.R. Wisehart, General Research Corporation, PO Box 3587, Santa Barbara CA 93105

IFTRAN-2 W.R. Wisehart (see IFTRAN-1)

IFTRAN-LA Martin Cohen, USC Information Sci Inst., 4676 Admiralty Way, Marina del Rey CA 90291

IFTRAN-C W.R. Bezanson, Carleton Univ., Ottawa Ontario, Canada

IFTRAN-W D.L. Dietmeyer, Univ. of Wisconsin, Madison WI 53706

STRAN W.L. Johnson, Rm 2218, Bldg 3, Ford Motor Co., PO Box 2053, Dearborn MI 48121

S-Fortran Guy de Balbine, Caine, Farber and Gordon Inc., 1000 E Walnut St., Pasadena CA 91106

STAPLE Selden L. Stewart, Adm A 221, National Bureau of Standards, Washington DC 20234

SFTRAN John Flynn, Jet Propulsion Lab, MS 125-128, Pasadena CA 91103

IPLFOR Arnold Schwartz, MS 168-427, Jet Propulsion Lab, Pasadena CA 91103

MORTRAN-1 A. James Cook, Computation Res. Gp. Stanford Linear Accelerator Center, PO Box 4349, Stanford CA 94305

MORTRAN-2 A. James Cook (see MORTRAN-1)

MORTRAN-X Richard H. Ault, MC 1760, Physics Dept, Univ. of Utah, Salt Lake City 84112

DEFT T.E. Hull, University of Toronto, Toronto, Ont. Canada

TRANSFOR Loren Carpenter, Boeing Computer Services, MS 73-13, Renton WA 98055

RATFOR Brian Kernighan, Bell Labs, Murray Hill NJ 07974

FLECS Terry Beyer, Univ. of Oregon, Eugene OR 97401

SPA S.A. Steele, 101-230, RCA, Moorestown NJ 08057

SFOR Dennis O'Neill, Bell Labs, Holmdel NJ 07733

SFP Robert Rich, Johns Hopkins Univ., Applied Physics Lab., Silver Spring MD 20910

SF-Visnavich E. Towster, Computer Sci Dept, PO Box 4330, USL Station, Lafayette LA 70501

WATFIV-S Paul Dirksen, University of Waterloo Waterloo, Ontario, Canada

S-WATFIV-Hisgen N.E. Gibbs, College of William & Mary, Williamsburg VA 23185

NSF-Tran Frank Friedman, Temple Univ., Philadelphia PA 19122

DO-OD IF-FI Structured Fortran J.L. Wagener, State Univ. of NY, Brockport NY 14420

LINUS J.D. Woolley, Dept. of Computer Sci., Bowling Green State Univ., OH 43403

HIGGINS D.S. Higgins, B-3, Florida Power Corp. PO Box 14042, St Petersburg FL 33733

MELTRAN J.S. Miller, Intermetrics Inc., 701 Concord Ave, Cambridge MA 02138

DAY-SMP A.C. Day, Computer Center, University College, 19 Gordon St., London WC1H OAH England

SFORTN L.L. Pooler, Hughes Aircraft, PO Box 1638, Oceanside CA 92054

PREFOR W.M. Bradley, MC 96, IBM Corp., 1322 Space Park, Houston TX 77050

SPDS Dick Gormley, IBM/FSD, 2625 Townsgate Rd. Westlake Village CA 91361

Fortran-4S R.S. O'Bryant, Texas Instruments, PO Box 6015, Dallas TX 75222

PSST Leon Stucki, A3373-B4DC MS-13-2, McDonnell Douglas, Huntington Beach CA 92647

SUGFOR Anders Beckman, Uppsala Univ., Uppsala, Sweden

SAFTE Lance Moberg, Sperry Univac, PO Box 3525 St. Paul MN 55101

ELESSAR Redford Bond, Essay Corp., 100 Park Ave., Oklahoma City OK 73102

FDP-5798-CDW Guy McCool, Dept. 84F, IBM Corp, 620 N Brand Blvd, Glendale CA 91203

List of Structured Fortran Processors, cont.

ATHENA-ML Ira C. Hanson, Dept 19-43, Bldg 201, Lockheed Palo Alto Res. Lab., Palo Alto CA 94304

SF-Henke W.L. Henke, MIT, Cambridge MA 02139

ESFOR W.L. Bearley, Citrus College, Azusa CA 91702

SHELTRAN G.A. Croes, Shell International Petroleum Ltd., Shell Centre, London SE1-7NA England

COM-SP R.E. Jeffries, COMSHARE Inc., PO Box 1588, Ann Arbor MI 48106

TR6B-TR7X L. Mossberg, Volvo Flyg. AB, S-461 01, Box 136, Trollhättan, Sweden

SPL&C A. Holgado, Univ. of Michigan, Ann Arbor MI 48106

B4Tran L.P. Meissner, Lawrence Lab, Berkeley CA 94720

SF-RIO R.N. Melo, Pontificia Univ Catolico, Rio de Janeiro Brazil

SF-Concordia T. Radhakrishnan, Computer Sci. Dept., Sir George Williams Campus, Concordia Univ., 1445 de Maisonnueve Blvd. West, Montreal, Quebec H3G 1M8, Canada

SF-CHAT Fred Foldvary, L-32, Lawrence Lab., PO Box 208, Livermore CA 94550

SFOR-LSI Tetsuo Mizoguchi, Electronics Lab, Mitsubishi Elec. Corp., 325 Kamimachiya, Kamakura, Japan 247

MTUFP J.Lowther, Computer Sci. Dept., Michigan Technological Univ., Houghton MI 49931

Total to date: 51 processors. Please report any additions to this list. (See article on The West Coast Working Group, page 2.)

Fortran Bibliography
(Installment no. 2: see also FOR-WORD Newsletter No. 2)

G.L. Perry and J.T. Sommerfeld, Fortran programming aids. Software Age, Oct-Nov 70

The next standard Fortran? (Report of a working party of the British Computer Society's specialist group on Fortran.) Computer Bulletin, Feb 71

A.H.J. Sale, The classification of Fortran statements. Computer Journal, Feb 71

D.E. Knuth, An empirical study of Fortran programs. Software P & E, Apr-Jun 71

D.D. McCracken and G.M. Weinberg, How to write a readable Fortran program. Datamation, Oct 72

F. Thorlin, What's new with DO? Datamation, Dec 72

F. Engel, Future Fortran development. SIGPLAN Notices, Mar 73

J. Larmouth, Serious Fortran. Software P & E, Apr-Jun 73 and Jul-Sep 73

D.C. Hoaglin, An analysis of ... Knuth's empirical study. Software P & E, Apr-Jun 73

D.D. McCracken, Is there a Fortran in your future? Datamation, May 73

P.B. Schneck and E. Angel, A Fortran to Fortran optimising compiler. Computer Journal, Nov 73

F. Engel, Revise standard Fortran? Datamation, May 74

C.W. Barth, Notes on the CASE statement. Software P & E, 1974

B.G. Ryder, The PFort verifier. Software P & E, Oct-Dec 74

L.P. Meissner, A method to expose the hidden structure of Fortran programs. Proc. ACM 74

D.E. Whitten and P.A.D. de Maine, A machine ... independent Fortran (PFortran). Software Engineering, Mar 75

P.M. Neely, After Fortran, what? (Guest editorial) Software P & E, 1975

W.F. Ross, Structured programming: highlights of the 1974 Lake Arrowhead workshop. Computer (IEEE Computer Society), Jun 75

See also SIGPLAN Notices, Mar 75, articles by C.A. Muntz; D.L. Presberg and N.W. Johnson; K.G. Stevens, Jr.; R.Zwakenburg, J. Engle, D. Gotthoffer, and M. River; D. Wedel; and H. Krohn. This is a special issue of SIGPLAN Notices, containing proceedings of a conference on programming languages for parallel and vector machines. Many of the articles are of interest to Fortran users and to developers of extensions to the Fortran language.

See also other issues of many of the journals listed in the references above.

Note

In the fall of 1974, Peter Brown wrote an entertaining series of articles for "Computing Europe" (a publication of the British Computer Society) on the theme, "Taking a stroll down Babel Street." He characterizes various computer languages as shops along a street, competing for customers. Each article is accompanied by a half-page illustration (suitable for framing). Fortran is featured in the issue of 31 October 74. Fred Fortran is characterized as the proprietor of a grocery store that carries few lines of merchandise, which remain unchanged for years. He still stocks "the old Brand 704 plain subscripts" and does not carry the fancy new-fangled varieties stocked by his competitors. When someone asks him to fill an order N times, he doesn't check to see if N is zero; he gets on with the filling of the order and checks afterward, since this procedure is more efficient. He accuses Lady Algol of spending so much time checking everything that she takes forever to get anything done. "He did not show any imagination, just thoroughness and efficiency, and the result was low prices and lots of customers."

On Fortran Syntax Analysis Methods

Attempts to use higher level languages for implementing Fortran compilers sometimes fall into difficulty because of the fact that blanks (spaces) in Fortran are generally not significant. As a result, lexical analysis cannot be done on the basis of a simple left-to-right scan. For instance, it is not until the comma is reached that the compiler can determine that "DO 7 I = 1, 5" is not an assignment to the variable DO7I.

One way out of this difficulty is to make a very quick preliminary scan of the source code to determine the statement type. Once this has been established, a left to right scan can be used for the detailed syntactic analysis. Preliminary scanners for statement recognition purposes have been reported by A.H.J. Sale (The classification of Fortran statements; Computer Journal, Feb 71), by Fred Crary (Math Research Center, University of Wisconsin), and by Don J. Orser (Applied Math Division, National Bureau of Standards). There are ways to make these pre-scanners very efficient, and even to collect additional information; for example, for each left parenthesis at the outermost level, the location within the statement of the matching right parenthesis.

This technique is no doubt well known to experienced compiler developers, but it may be of interest to others who are interested in adapting to Fortran the methods they have developed for other language processors.

CORRESPONDENCE

The following are excerpts from letters received in response to previous FOR-WORD Newsletters and to announcements from the West Coast Working Group on Structured Fortran.

Best Wishes

** I would be interested in following the progress of your committee to make Fortran amenable to structured programming. I am a user of Fortran in the areas of real-time industrial and engineering programs. If I can be of assistance to your committee, please let me know.
Stephen Kessler, San Jose CA

** I am becoming a more and more enthusiastic supporter of Structured Fortran.
Ben Shneiderman, Bloomington IN

** I certainly support the activities of the West Coast Structured Fortran Working Group and would be happy to have you suggest ways in which I and the group of people working with me here might participate.

The areas in which your group intends to work seem quite appropriate and, in general, complementary to the things we are doing here. Our current efforts are aimed at developing a general syntactic extension to Fortran. It will certainly be most valuable to us to have as much input as possible from others. We are not optimistic that much can be accomplished through ANSI but there is certainly no reason not to try.
Anthony Ralston, Buffalo NY

** I am interested in participating in whatever ways might be useful to the Working Group and to this Laboratory's future programming effectiveness. I recently gave a talk to Laboratory members on things like structured programming, correctness proofs, etc., and some interest was expressed in actually acquiring or developing a "structured Fortran" and making a serious attempt to make it a standard software production tool at this installation.
Erich Knöbil, Ithaca NY

** I am interested to see the formation of the Structured Fortran Working Group, and am strongly in favor of the formation of a SIGPLAN Special Technical Committee on Fortran. It would seem that such a group would correspond fairly closely (in structure and objectives) to the British Computer Society's Fortran Specialist Group, which was formed about three years ago and now represents an authoritative voice on Fortran matters for the B.C.S.
Alan Clark, Herpenden, Herts.
(Secretary, BCS Fortran
Specialist Group)

CORRESPONDENCE, cont.

** My position is that of supporting a large (50 to 60) population of Engineers who do modelling in Fortran, and doing whatever independent programming I can find time for. The environment is such that Fortran is the only choice currently, and for some time to come.

My selfish interests in the working group are picking the brains of others in similar positions as myself. My less selfish interest is a belief that Fortran is basically a good language which needs some extensions to prevent the ugly code which is daily brought into my office."

Larry Babb, Mountain View CA

Teaching Structured Programming to Fortran Users

** My own job is teaching. Most of my students are open shop programmers who haven't learned any language but Fortran, and for the most part it's very hard for them to accept the idea of writing in a pseudo language. In my program design course we spend five two-hour lecture periods on program readability, software vs. hardware cost, modularity, top-down vs. bottom-up programming, program correctness, etc. before we're in a position to talk about block structure and the need for a pseudo-language. My "Suggested Steps for Writing a Computer Program" is handed out even later in the course. In it I've tried to describe the ideas and the approach used by our most successful local programming group. By that time, most of the students are at least able to see the rationale for structured programming, even if they don't agree with it. But there's always someone who says, "Why go to all that trouble when all you want to do is write a Fortran program in the first place?"

Ted Tenny, Sunnyvale CA

Productivity

** We have been experiencing this "Structured Fortran" for more than one year. We don't think that programming productivity grew significantly, but we think it is worth while because two persons are working on one project at same time with different skill or knowledge.

Tetsuo Mizoguchi, Kamakura, Japan

All Fortran control structures are illegal.

** There are major differences between SHELTRAN and other approaches that we have seen. Statement labels (except for Format) are illegal, and hence also all Fortran control structures. This was done in order to enforce a top-down design and programming style and has done wonders. There were no complaints and virtually no training was required. A great emphasis was put on readability and hence documentation.

G.A. Croes, London, England

Is Preprocessing the answer?

** I cannot help but put in a "second" for the note, "On Preprocessors (Editorial)" in Newsletter No. 3. As a developer of programs for paying customers who want ANSI Fortran programs as a deliverable end-item, I am very concerned about what preprocessors produce. My use of a preprocessor is predicated on this point. Preprocessors -- at least the one I am using -- have justified themselves to me as a time saver in several ways. Principally, development of the code proceeds faster and debugging is eased -- especially that part involved in the translation of "meta-code" or design language to Fortran.

I am, by the way, a recent convert to the preprocessor idea as my Fortran is typically tightly structured with a minimum of labels and very systematic use of GO TO's. (One learns these things from bitter experience!)

Richard Swanson, North Bend WA

** I am a Fortran programmer, but I recognize its current limitations with respect to structured programming, string manipulation, recursion, etc. I am interested in extensions to Fortran which enhance its capabilities and widen the applications for what is useful.

Unfortunately, I do not consider preprocessors anything but a stop-gap measure -- at best a chance to try out various new features, and at worst an illustration that no one is serious about extending (or better yet, modernizing) Fortran so that it is applicable outside the university/research environment.

In my opinion, what is needed is: (a) a formal description of a modern Fortran where the additions -- control statements, new syntax / semantics -- is carefully thought out. (b) Several working compilers (Fortran to object code) for selected machines. These must produce object code which is reasonably compatible with that produced by current compilers. (c) A concerted series of presentations, papers, etc. on (the modern) Fortran. Until a compiler is available for a properly defined Fortran for several significant computers, I think this subject will be regarded as mostly a curiosity, to be ignored, except for its (limited and scattered) supporters.

Douglas Whitten, University Park PA

** I feel very strongly that Fortran language extensions should be implemented directly into the compiler and (that preprocessors) are too cumbersome and inefficient to be effectively used in very large computing installations. Also I have observed several precompilers which generated awful Fortran code.

CORRESPONDENCE, cont.

Portability of programs is a very desirable goal but going from one dialect of Fortran to another is only the tip of the iceberg of problems. Portability depends heavily upon the hardware, arithmetic structure and job control language. With such diversity of existing systems, portability is best controlled with "good programming practices." Write the program in a clear simple fashion with adequate comments. It will then be translatable to the receiving system with a minimum of effort.

Ira Hanson, Palo Alto CA

** (Concerning the survey being undertaken by the West Coast Working Group on Structured Fortran:) I suggest establishing a clear distinction between the language extensions that are being proposed and the implementations of these extensions. There are amateurish implementations of good language ideas. Should they be ignored because they are not properly supported? Conversely, there are also some suggestions that appear to be unreasonable within the working time frame. Nevertheless, they should be recorded and commented upon. Any known tool whose purpose is to allow structured programming in a Fortran environment should be documented. Please, let us not limit the survey to a mediocre solution to the problem, namely the Fortran preprocessor. Such an obvious bias at the data collection stage can only reduce the power and credibility of the conclusions and recommendations that we may derive from this work.

Guy de Balbine, Pasadena CA

Masters Thesis on Structured Fortran

** I am a student of computer science at the University of Tampere. I am finishing my studies and making my master thesis. The theme of my thesis will be the structured programming especially extensions to Fortran which support structured programming. In the first part I review some existing extensions to the Fortran language and in the second part I will introduce my own version of "extended Fortran."

Jorma Siintoharju, Tampere, Finland

Structured WATFIV

** We have modified WATFIV to include several new statements and a structure so that one can do Structured Programming in Fortran. I should mention that the particular constructs chosen by us were governed to a large part as to make the implementation easy. When we finally started modifying the compiler, it took us one week to do the necessary coding and another week to test our changes.

Paul Dirksen, Waterloo, Ontario

Implementation problems

** I am implementing a Fortran code generator, where the input is a description of the program control flow and a sequential file of Fortran text data. While working on the code generator I ran into some interesting problems: Arithmetical IF; Boolean IF, reversing the Boolean expressions or generating a GO TO in front of the THEN block or reversing the THEN and ELSE blocks; Pre-checked DO loop, if the start and /or stop value are variables, a test can be made before entering the DO-block; Break (exit or escape) out of loops, if an escape is wanted out of nested loops, the escape must be qualified; Avoiding the generation of CONTINUE as much as possible. I would like to discuss these problems with you and get your opinion about them. Did you have the same or similar problems and how did you manage to solve or avoid them?

Klaus Kirchhof, Munich, Germany

Compiler standards needed

** Thank you for your letter regarding structured Fortran. The process of which it is a part is of great interest to me. Possibly severe regulation of the use of COMMON will be considered an aspect of (use of the existing Fortran language to write structured programs). Many users at the Indian Institute of Technology, Kanpur, have been driven to using COMMON simply because the Fortran compiler there imposed a limit of 300 references to subprogram arguments! The programming rules then should be linked with compiler standards which make their use feasible.

Hari Sahasrabuddhe, Waterloo, Ontario

Source program format

** I would like to advance the ideas of multiple statements per card and automatic formatting of source listings. The automatic indenting is almost as valuable as the language extensions.

D.L. Dietmeyer, Madison WI

Text substitution

** I draw to your attention a feature of the Burroughs B6700 / 7700 Algol language, the DEFINE declaration (text substitution), and suggest that a feature of this kind added to the Fortran language might assist those interested in its development to experiment with structured forms. Extracts from Burroughs reference manual are attached.

B.A.M. Moon, Christchurch,
New Zealand

CORRESPONDENCE, cont.

Statement grouping

** I have not followed the common approach of trying to emulate Algol, because I don't consider Algol to have the best control structure features, and also because I wanted to make the implementation of a preprocessor as simple as possible. The key element in my system is the use of block identifiers, which are not merely entry-point labels, as in Algol, but define the entire block. Once one has such identifiers, then several other useful structure elements become feasible.

EXIT has been demonstrated to be necessary in structured programs. The "EXIT blockname" enables exiting any number of nested levels, and therefore helps in writing clear programs without using unnecessary control flags, as is needed by DO-WHILE. That is why I avoided DO-WHILE -- it is not good for top-down programming, because it requires the initialization of control variables external and prior to the block in which they have a meaning.

The specification for any structured Fortran should be such that ordinary Fortran statements can be intermixed with the structured Fortran, so that existing programs can be amended using the structured notation, without having to totally rewrite.

Sylvan Rubin, Palo Alto CA

** I think you are right when you write that Algol-like ways of grouping statements (begin ... end) are not well suited from the point of view of extending existing Fortran compilers.

Bertrand Meyer, Clamart, France

** The change I advocate most and the one I believe will make the most improvement for the size of the change suggested is to add "DO TO n", a particularly useful construction on the end of an IF condition.

Perhaps the second most needed change would be a read / write core under format control. The third change I would suggest is in the area of string handling.

In the area of conveniences, Fortran would be improved if it had vector and matrix operations, and the ability to specify action on error, such as divide by zero.

R.A. Baker, Houston TX

Data structures

** What happened to data structures? Existing macro preprocessors implement in one form or another the control statements appropriate to structured programming. They do not, in general, address the problem of adequate data structuring facilities, so the usefulness of the languages currently implemented for serious structured programming is only partial. Happily, we can implement at least the records and references of Algol-W, Pascal, etc. in a restricted form, using Fortran arrays. This has been accomplished via the table-driven macro translator for the MORTRAN-2 language, but the implementation technique can presumably be incorproated into other macro-preprocessors. Implementation of stacks and queues is very simple also.

THe ideal situation would be a flexible (presumably table driven) yet reasonably efficient macro-translator existing as a standard Fortran program, and a "core language" implemented by this translator into standard Fortran. This core language would presumably contain the normal control statements associated with structured control; some form of implementation of records and references should also be included.

Charles Zahn, Stanford CA

** I want to get my two cents worth into a definition of a Fortran of tomorrow, and want to lay out here where my specific interests lie.

I'm sure you people have structured programming concepts well in hand. My interest is more in the area of data definition, on which I have some rather strong ideas. For instance, it ought to be possible to declare REAL numbers in terms of their number of digits of significance. Another need would be to define data widths in terms of how many characters a variable can hold -- e.g. "CHAR * 30 NAME", NAME then occupying 3 words on the 7600 and 8 words on the 370.

I've been involved in writing pattern recognition programs in Fortran over the last few years, which is an activity that leads one to wish desperately for a facility like the PL/I BASED variable -- i.e., a multi-word data object that is referenced via a pointer word. I think such a thing could be crammed into Fortran without too much violence, by making it look like a Labelled Common block -- e.g., "BASED /BLAH/ A, B, C" and then some way has to be invented of shoehorning explicit or implicit pointer-qualifications into the language. There are many possibilities. I'm not asking for actual verbs like ALLOCATE and FREE, just the possibility of doing one's own allocation and garbage collection out in Blank Common and then easily referring to the created structured data.

Ed Fourt, Berkeley CA

CORRESPONDENCE, cont.

General suggestions for Fortran development

** For old Algol users the drawbacks of Fortran were obvious. Here are some: (1) The lack of control structures; (2) the lack of data structures (we used an Algol dialect called Algol-Genius, the data structures of which are inspired by Cobol); (3) The lack of blocking facilities and the use of internal variables within blocks; (4) The fact that most Fortran compilers, using part compiling, are not able to detect as many errors at compile time as an Algol compiler, for example to check the number and types of the actual arguments in a subroutine call.

Lars Mossberg, Trollhättan, Sweden

** Personally, I feel that a non-recursive Algol with Fortran's input-output characteristics (including mechanisms for direct access files) would be an extremely powerful language that would be easy to implement and learn. This obviously does not take into account the inertia of current Fortran implementation and applications, and I don't seriously propose it. I use it simply to convey to you my current view of the language I would like to be using.

Warren Smith, Sacramento CA

** It is an unfortunate fact of life that Fortran and Cobol will certainly be with us for a while and if the techniques of structured programming are ever to receive mass acceptance, they must be made easily available within these languages.

Just as structured programming is more than GOTO-lessness, so a good preprocessor should provide more than just extensive control structures. Provably the most important feature needed is a provision for local subroutines.

I have long felt that one of the great deficiencies of Fortran for doing reliable programming was the six character limitation on variable names.

An INCLUDE facility for copying in arbitrary program text would allow standard code, COMMON definitions, and the like to be entered in many places identically. Simple macro processors which offer no more than a straightforward string replacement aid in the maintainability and portability of programs.

Most preprocessors that I have seen provide free format input and embedded comments.

Wrandle Barth, Greenbelt MD

** This letter is written in response to the request of the committee for information about structured Fortran precompiler implementations. Its purpose is to describe a language, ELESSAR, which may be of interest. A note is to appear in SIGPLAN Notices as a description of the language. I will amplify the note in order to point up the features of this language that I think the committee should consider.

1. Free form. I think it is necessary for a structured language to have free format. One way to implement free form is (a) let the structured language keywords appear anywhere on a card and (b) use a statement separator between Fortran statements, as has been done for many years in timesharing compilers.

2. New language. If a structured Fortran is presented as a set of extensions to Fortran, it is to be expected that the user will have to learn a set of restrictions applicable to each extension. If it is presented as a new language, the entire language can be described to a Fortran programmer in very little space and with very little burden on the user's memory. The key to this is the fact that, without exception, "any Fortran statement is an ELESSAR construct."

3. Keyword lexicography. After a great deal of thought, I have come to the conclusion that the use of a special character as part of each keyword is justified.

4. Statement brackets. Regardless of what statement brackets are adopted as standard, I strongly recommend that the ending keyword somehow reflect back to the beginning keyword.

5. Spaces. Spaces should be significant.

6. Control constructs. I assume that there is little argument with the necessity for the simple WHILE, REPEAT and IF constructs, although I realize that Meissner has advocated retaining the flavor of Fortran, a position with which I strongly disagree. I also suppose there is little argument with the desirability of some sort of an internal subroutine capability. I recommend that the internal subroutine not contain a RETURN statement; this would be a temptation for a programmer to use mutiple exits. I strongly recommend REPEAT ... UNTIL ... DO ... END (REP). Since implementating this precompiler, I have considered more than a hundred possible ways to implement the n-and-a-half repetition loop. Zahn's event indicators are hard to learn, unnatural and rigid. Dahl's proposal is also unnatural, even without the semicolons and colons which he proposes. LEAVE or EXIT statements have a "jumpy" connotation. If I say in English "Repeat your exercises until sundown" or "Until sundown do pushups" I believe it is natural and clear.

7. Speed. If a precompiler is to appeal

CORRESPONDENCE, cont.

to a broad class of users, it must be fast.

8. Parsing. Any standard must require some sort of simplified parse of the structured language code. I realize that it is only necessary to recognize keywords in order to translate correct code but decent error recognition and recovery cannot be accomplished unless the translator knows what it is doing.

9. Comments and CONTINUEs. I recommend a precompiler which does not copy comments from the source structured code and does not insert any comments. I think it is desirable to produce Fortran code that is as unreadable as possible in order that users will not be tempted to tinker with the Fortran code but will instead make changes in the structured language code. Also the preprocessor output should contain as few CONTINUE statements as possible.

10. Portability. ELESSAR will translate any Fortran. This is because of its syntax.

This precompiler has all the control structures which I need or would like to have (except for possible deficiencies in the CASE statement), and there are no redundant features except possibly the FOR and the WHILE ... DO ... UNTIL. Its efficiency is due to the fact that non-portable input and output routines are included.

I do not believe that a standard can win acceptance if it is only a Fortran-flavored set of extensions.

I will be happy to furnish additional information concerning the details of the implementation and the techniques employed in the parse, the semantics and the elimination of CONTINUEs.

Redford Bond
Essay Corp, 100 Park Ave.
Oklahoma City OK 73102

(Ed. note: this letter has been condensed.)

Mailing list changes; copies of previous FOR-WORD Newsletters

Please use the bottom half of this page, or a photo-copy, to report mailing list changes or to request copies of previous FOR-WORD Newsletters.

( ) I enclose mailing label from FOR-WORD Newsletter No. 4 (or I am using entire bottom half of page, containing mailing label on the outside). Please change mailing address to that shown:
( ) Add the following name and address:

_____

_____

_____

_____

Please send me a copy of FOR-WORD Newsletter:
( ) No. 1 ( ) No. 2 ( ) No. 3

FOLD PAGE IN FOURTHS AND MAIL

TO:
Loren P. Meissner (50-B 3239)
Lawrence Berkeley Laboratory
University of California
Berkeley CA 94720

= F O R - W O R D = >

FORTRAN DEVELOPMENT NEWSLETTER

October 1975 -- No. 5 -- Published by
Ad Hoc Committee on Fortran Development
ACM-SIGPLAN

Table of Contents

CONCERNING FOR-WORD

This Newsletter is an informal publication of the SIGPLAN Fortran Development Committee. This is an ad hoc committee created by SIGPLAN for the purpose of taking steps toward formation of a Special Technical Committee on Fortran Development. Interim officers are Paul B. Schneck, New York NY (Chairman); Guy J. de Balbine, Pasadena CA (Vice Chairman); Michael A. Malcolm, Waterloo Ont (Secretary-Treasurer); and Loren P. Meissner, Berkeley CA (Editor). Further information concerning the purposes of the committee may be found in previous issues of FOR-WORD.

The Newsletter mailing list will eventually form the basis for a Membership List to be maintained by ACM. It is expected that, after a period of time during which activities of the group are supported by SIGPLAN, it will eventually be necessary to charge a membership fee to cover the cost of distributing the Newsletter.

Please send mailing list changes (with a photocopy of mailing label, if possible) and additions to the Editor at the address given below.

Copies of back issues of FOR-WORD are also available from the Editor.

Contributions solicited. Please send items of interest to the Editor. Especially solicited are letters or articles concerning the directions that future Fortran development should take. Concise reviews of new books or articles pertinent to Fortran language development or modern Fortran programming methodology may also be contributed. The Editor reserves the right to excerpt all correspondence.

Mailing address:

Loren P. Meissner
50-B 3239
Lawrence Berkeley Laboratory
Berkeley CA 94720

CALENDAR

2-5 Dec 1975	Fall DECUS U.S. Symposium (Digital Equipment Users Society), Los Angeles CA. Chairman: William J. Lennon, Northwestern University.
19-21 Jan 76	Symp. on Principles of Programming Lang., Atlanta GA. Program Chmn: S.L. Graham, U of CA, Berk.
2 Feb 76	Fortran Specialist Group, British Computer Society. London. *Visitors to England are welcome.* P.A. Clarke, Rothamsted Exp. Sta., Harpenden, Hertfordshire.
9 Feb 76	West Coast Fortran Forum (see notice below).
10-12 Feb 76	Computer Science Conf 76, Anaheim CA. Chairman: J. Feldman, U of CA, Irvine.

Forthcoming meetings of ANSI X3J3:
8-12 Dec 1976, Marlboro Mass.
19 Jan or 2 Feb, Southern Calif.

THE FORTRAN FORUM IS COMING

The Los Angeles Chapter of the ACM and the National Bureau of Standards, in cooperation with the ACM (SIGPLAN) and the IEEE Computer Society, is sponsoring the West Coast Fortran Forum to be held on Monday, February 9, 1976, at the Disneyland Hotel -- the day before the ACM Computer Science Conference. At the Forum there will be discussions of the soon-to-be-released draft proposed American National Standard X3.9 Fortran revision. Representatives from the Fortran Standards Committee will be available to answer questions and clarify concepts. Chairman: Donald Reifer, PO Box 92957, Los Angeles CA 90009

FORTRAN STANDARDS ACTIVITY

X3J3 Committee Activities

The ANSI Fortran Standards Committee (X3J3) has prepared a document for letter ballot. This document, if approved, will become the basis for further action by ANSI, including possible publication for public review and comment as a draft proposed American National Standard Fortran (revised).

The document describes a full language and a subset. The criteria for the subset include "a minimum demand on storage requirements, particularly during execution," and "a minimum of effort for the development and maintenance of a viable FORTRAN processor."

A more inclusive subset has also been under consideration by the committee; however present plans are to forego inclusion of the intermediate subset in the present document, but to continue work on it as a recommendation to be released later.

Progress on a new Standard Fortran, by Walter S. Brainerd, Pasadena CA

[The following notes were prepared for an informal session that was held to discuss progress on the proposed new standard.]

During preparation of a proposed new standard Fortran, the following changes have been voted by ANSI X3J3 through August 1975. Only significant changes are listed here; many additional minor changes have also been approved.

1. The main program may contain a PROGRAM statement. A BLOCK DATA subprogram may have a name.

2. Character data type:

   a. Constants: 'ABC'

   b. Declaration of character variables and arrays:

```
CHARACTER*4  A, B*8, C(9, 9)*5, D
```

   c. Concatenation operator: B // 'Q'

   d. Substrings:

```
B(2:5) = C(3, 5)(3:4) // D(2:3)
```

3. Expressions:

   a. Integer, real, or double precision expressions may appear in subscripts, DO parameters, computed GO TO, etc.

   b. Expressions of all types may appear in output lists.

   c. Character expressions may be used as format specification.

```
PRINT '(I3, A1, I3)', I, '=', J * K
READ (5, '(A80)') CHARS
```

   d. Mixing of arithmetic data types is permitted.

   e. The processor may evaluate any equivalent expression, except that integer division and integrity of parentheses must be preserved.

4. Implicit types of names beginning with designated letters may be declared by means of IMPLICIT statement. Also character lengths.

5. PARAMETER statement:

```
PARAMETER N = 1.7, C = '((('
```

   a. The type of the name depends upon the constant.

   b. A parameter name may appear as a primary in an expression, in a data statement, and as the character length in an IMPLICIT statement.

6. Arrays:

   a. Arrays may have as many as 7 dimensions.

   b. Explicit lower bounds for arrays (with lower bound default = 1):

```
REAL A(-3:-1, 0:7)
```

   c. Adjustable dimensions may be transmitted via COMMON as well as in the argument list.

7. Computed GO TO defaults to following statement if the value of the control expression is out of range.

8. DO loops:

   a. Parameters, which may be integer, real, or double precision, are evaluated and converted to the type of the DO variable.

   b. The increment value may be negative.

   c. A trip count is established initially, and is unaffected by changing entities in parameter expressions. The DO variable must not be changed during execution of the loop.

   d. The minimum trip count is zero.

   e. The DO variable remains defined at completion, with the value it would have had if the loop had been executed one more time.

   f. Examples:

```
      Q = .1
      DO 9 X = .9, Q / 2, -Q
         A = X
    9    Q = 2 * Q
```

The loop is executed 9 times; at completion, A = .1, X = 0.

```
      N = 1
      DO 9 I = 5, 3
9         N = N - 1
```

The loop is executed 0 times; at completion, N = 1, I = 5.

9. Keyword specification of unit and format in input-output statements; also error and end of file specifiers:

```
END = specifier (statement label)
ERR = specifier (statement label)
UNIT = specifier (expression)
FMT = specifier (reference to statement label or character entity)

READ (ERR = 99, UNIT = N - 2, END = 98,
1  FMT = F) A
```

10. Stream input-output: the property is specified for a file. The next character read or written is the one following the last one read or written, regardless of record boundaries.

11. Direct-access input-output: the property is specified for a file. An input-output statement on a direct-access file transmits one record.

```
WRITE (9, REC = K * 2) B
```

12. Internal files: a character entity name may be used in place of a file name for an input or output statement. (This implements the features of ENCODE and DECODE.)

```
CHARACTER*5  C
WRITE (C, 9) I
9  FORMAT (I 5)
```

13. OPEN statement: OPEN (list)

The list must specify a unit number, and may also include any of the following:

```
ERR = label
NAME = file name (e.g., 'ABC/DEF')
STATUS = 'OLD'
         'NEW'
         'SCRATCH'
         'UNKNOWN'
ACCESS = 'SEQUENTIAL'
         'STREAM'
         'DIRECT'
FORM = 'FORMATTED'
       'UNFORMATTED'
RECL = record length in a direct access file
MAXREC = largest record number in a direct access file
```

14. CLOSE statement:

```
ERR = label
STATUS = 'KEEP'
         'DELETE'
```

15. INQUIRE statement:

```
UNIT = or FILE =

ERR = label
EXIST = logical
OPENED = logical
NUMBER = integer (number of unit connected to file)
NAMED = logical
NAME = character
ACCESS = character ('SEQUENTIAL', 'STREAM', or 'DIRECT')
FORM = character ('FORMATTED' or 'UNFORMATTED')
RECL = integer
MAXREC = integer
```

The INQUIRE statement permits programmatic determination of the properties of a file or unit.

16. A FORMAT statement label reference may be set by an ASSIGN statement.

17. New edit descriptors:

I *w.m*	at least *m* digits (left zero fill)
E *w.d* E*e*	*e* digits in exponent field
E *w.d* D*e*	*e* digits in exponent field
A *w*	character data
A	character data (field width determined by character entity)
'XXXX'	character constant (output only)
T *n*	tab to position *n*
±*n* X	relative tab of *n* positions (right or left)
:	terminate input-output if list is exhausted
±S, S	control of optional plus sign
BZ, BN	input blanks converted to zeros or skipped

18. List-directed input-output: "*" in place of format identifier specifies a "default" format determined by the input-output list items, and on input by the form of the data.

```
READ *, I, A(I)
```

19. Intrinsic functions now include those formerly called intrinsic and basic external functions.

a. Generic functions. Most of them return a value having the same type as the argument(s). For example,

```
SIN (2.3D2) is double
MIN (1.7, X) is real
```

b. Type conversion generic functions may have integer, real, double, or complex arguments:

INT
REAL
DBLE
CMPLX

For example, REAL when applied to integer arguments performs FLOAT operation; for double precision arguments truncates to single precision; and delivers real part of complex arguments.

c. CMPLX and ATAN may have one or two arguments.

20. The INTRINSIC statement allows an intrinsic function to be passed as an actual argument.

21. The EXTERNAL statement identifies an external procedure (not an intrinsic) and allows it to be passed as an actual argument.

22. The ENTRY statement allows alternate entry into a subprogram.

23. The SAVE statement specifies local variables and common blocks (not references to dummy arguments) to be saved between executions of a subroutine.

24. Alternate return:

Calling program:

```
        CALL SUBRTN (A, *14, B, *9)
        ...
9       CONTINUE
        ...
14      CONTINUE
        ...
```

Subroutine:

```
        SUBROUTINE SUBRTN (X, *, Y, *)
        N = 3
        RETURN N - 1
        END
```

Control is returned to the statement with label 9 in the calling program. RETURN N would be a normal return if N is less than one or greater than two.

Conflicts with X3.9-1966:

1. The Hollerith data type has been deleted. The *n*H edit descriptor in FORMAT statements has been retained, however.

2. The names of new intrinsic functions may conflict with programmer-provided external functions.

3. An intrinsic function that is passed as an actual argument must appear in an INTRINSIC statement.

4. Reading into an *n*H edit descriptor is prohibited.

5. The value of a subscript expression must not exceed the declared bound. Thus, neither of the following programs is permitted:

```
REAL A(2, 2)          SUBROUTINE SIGH (B)
A(3, 1) = 3.1         REAL B(1)
STOP                  B(2) = 2.2
END                   RETURN
                      END
```

6. In an EQUIVALENCE statement, an array must either have no subscript or have the same number of subscripts as declared dimensions. (Formerly, a single subscript was permitted for multiply dimensioned arrays.)

7. Mixing formatted and unformatted records in the same file is not permitted.

8. An input-output list or sublist must not be enclosed in parentheses.

9. Redundant type specifications are prohibited.

Some suggestions for minimizing difficulty with portability

1. Conversion of a real or double precision datum to an integer datum may not produce consistent results due to roundoff error. The following are standard, but not recommended, statements:

```
DO 9 X = .1, .9, .1
READ (UNIT = SIN (X)) A
GO TO (10, 20, 30) T * T - 7.3
A(X) = 0.0
```

2. No collating sequence for the characters is specified, except that the letters A to Z are in order and the digits 0 to 9 are in order. In particular, the relation of the blank character to the letters and digits is not specified. Thus the use of relational operators other than .EQ. and .NE. to compare character entities may not be portable.

3. A function will not necessarily be evaluated, if the value of the expression in which it appears can be established without its evaluation. Thus, if the function has side

effects (e.g., if it changes the value of an argument, creates output, etc.), these may or may not occur, depending upon the processor.

4. Each external function referenced in a program should appear in an EXTERNAL statement. Each intrinsic function that is not in the given Table should appear in an INTRINSIC statement in any program unit that references it.

5. Characters not in the Fortran character set should not be used.

6. The set of file names is processor dependent.

7. Use E *w.d* E*e* or E *w.d* D*e* edit descriptors instead of E *w.d* and D *w.d* edit descriptors.

Progress toward a Fortran Data Base Facility

Chester M. Smith, Jr., the chairman of the Codasyl committee on Fortran Database Manipulation Language, reports concerning progress on a data manipulation proposal for Fortran:

"We have a version, currently called Version 0.2, of what we hope will be a reasonable Journal of Development for a Fortran Data Base Facility. It is good in places and in others there are gaping holes. It is seventy pages in length and outlines all the major sections we are thinking about. Now some of the sections only consist of a section head, and others need much work, so I would prefer to keep the distribution down to a minimum. If on the other hand you feel the need for a copy, and I am sure that a number of people do, if you will write me a letter I will send you one."

Address: Chester M. Smith, Jr.
Computation Center
The Pennsylvania State University
University Park PA 16802

Onward toward Fortran-1984!

At the recent meeting of ANSI X3J3, an ad hoc committee on future revisions of the Fortran standards was appointed. This committee will make recommendations to X3J3 in several areas, including criteria for determining whether or not further revision (beyond the current efforts of the committee) should be undertaken, and if so, objectives and target dates for such revision, and criteria for proposed changes, as well as philosophy for defining subsets (proper subsets or modules, etc.) in a future revision. The ad hoc committee will also collect and classify features that have been proposed or that are suggested to the full committee in the near future, but which are postponed by the full committee for consideration as part of a possible future revision.

Members of the ad hoc committee are W.S. Brainerd (chairman), J.H. Matheny, L.P. Meissner, J.C. Noll, M.A. Rainer, and R.T. Slavinski.

Recent changes to Subset Fortran

At the recent meeting of X3J3 (Sunnyvale CA, 7 - 9 October 1975) it was voted to add the following features to the Fortran subset:

Character assignment and relational operators where the two sides are character entities of different lengths

The BN and BZ edit descriptors

The *n*H edit descriptor in a format specified by a character entity

Asterisk as well as C in column one of a comment line

It was also voted not to require asterisk fill in a real output field where the magnitude of the printed exponent exceeds 999; and not to incorporate the "short form" READ, WRITE, and PRINT statements (with no unit number specified) at the subset level. Character entities of different lengths will also be allowed in a DATA statement (i.e., the specified constant need not have the same length as the character entity to which it is pre-assigned.)

Some of these changes were already incorporated in the preliminary subset working document recently distributed by the committee on a limited basis.

Plans for distributing proposed Revision

The committee (X3J3) has voted to submit to its 20 or 25 members a letter ballot on the question of whether the current draft document should be forwarded as a "draft proposed American National Standard" revising X3.9-1966 Fortran. If the current document (which describes Fortran and Subset Fortran) is considered acceptable by the committee, it will be forwarded to X3 for further processing. If the current document is not acceptable, some later document will presumably be forwarded to X3. The first step to be taken by X3 will be to approve (or disapprove) release of the document for public review and comment. Current, most optimistic estimates are that release for publication could occur as soon as early February 1976.

ACM has offered to assist with the dissemination of this document as soon as it becomes available. It is planned to publish the entire document in SIGPLAN Notices, and to publish one or more summary articles in other publications of the ACM. (Hint: Non-members of SIGPLAN may have to purchase copies at a price nearly equal to the annual SIGPLAN membership fee.)

CORRESPONDENCE

On the Fate of Fortran

Robert F. Rosin, Ames IA:

I was pleased to find myself on the mailing list for FOR-WORD because, as I expressed when I met its editor in Los Angeles, I am curious about the fate of Fortran and how the competing forces of awakened enlightened programmers, harrassed threatened implementors, and conservative budget-conscious manufacturers will settle their differences.

But I was far less happy when I saw the statement "Fortran is dead" attributed to me completely out of context and without explanation (on page 5 of issue number 4).

In looking back at the short article from which I was quoted I found the following paragraph. "In my remarks at Houston, I said that I wished to declare that FORTRAN IS DEAD. Of course it is not dead in the literal sense; it is thriving. My claim, however, is:

1. Intellectually, it is dead, and
2. From a pedagogical point of view, its use in an introductory course tends to propagate its intellectual moribundity."

I thought at first not to write in response to this incident, for to do so might be interpreted as lending my support to the massive efforts to revitalize Fortran. In retrospect, however, I recognize that the majority of the people involved are either 1) from disciplines other than computer science and merely trying to make more useful the only programming language which they are accustomed to using, or 2) computer scientists who (perhaps because of hard economic times) are in a position in which they must use Fortran and are doing everything they can to overcome its obvious deficiencies. Therefore, I feel that it might not be inappropriate for a computer scientist, who is not committed to use Fortran, to use this occasion to offer a few observations to the readership of FOR-WORD.

My experience during the five years since my previously quoted statement appeared suggests that it remains as appropriate today, as it was then. It has always been my perception that Fortran was little more than machine language for the IBM 704 computer, with some syntactic sugar sprinkled over the top. The arithmetic IF statement, the plunge-ahead DO statement, variable names of 6 or fewer characters, limited character set, seriously restricted subscript expressions, and perhaps a few other Fortran fundamentals are archeological evidence of that extinct machine. Like so many others, I had my first "real" programming experience on the 704, but one must learn to recognize the changes which have taken place since its introduction in 1956. By the way, one can also recall that it took quite a while to convince a large part of the community of "scientific computer users" to program in Fortran rather than assembly language.

The letters published in FOR-WORD offer an overwhelming variety of suggestions for changing Fortran. It is interesting to note that many of them refer to "structured programming" as though there exists a wide-spread understanding of that term. To many people, structured programming does not represent a defined body of knowledge, but rather it is an on-going study which has no obvious conclusion in sight. Perhaps some of the most intriguing work in this field is being done by David Parnas at Darmstadt University in Germany. In his publications and manuscripts he generally indicates that his research into the production of quality software, through the application of the concepts of specifications and modules, is yielding results, and yet it presently offers very few language constructs, techniques, or rules which can be applied universally. (One can refer to his papers in the May and December, 1972, issues of CACM and in the proceedings of Conference of Reliable Software, in SIGPLAN Notices, June, 1975). The published work of others, such as Hoare and Liskov, support the contribution of concepts about data accessing and modules in this context. In contrast to this attitude of skepticism, I am disturbed to see strong implications, and even some direct statements, in FOR-WORD which indicate that, for example the addition of IF-THEN-ELSE and WHILE-DO to Fortran will allow the creation of structured programs in some automatic way.

Almost all of the suggestions one reads in FOR-WORD are for features to be added (or tacked on) to the language, and very few are for things to be taken away. One might suspect that changing the Fortran language in these ways might very well compromise two of the attributes which has made it so attractive over the past 18 (!) years; they are, fast compliation and fast execution based on having a relatively simple minded language to process. In order to preserve either or both of these characteristics it might be necessary to keep the language "small" by eliminating a number of constructs. (Although one can argue that preprocessors have a place in this scheme, they become expensive when used for the numbers of programs encountered in introductory courses in universities. I say this in spite of the fact that my own Fortran-to-MAD preprocessor, written in 1960-61, was used at Project MAC for several years.)

Of course deleting such goodies as the arithmetic-IF statement would lead to incompatibilities with the existing Fortran language. Apparently it is inconceivable to many people that an installation could have two compilers for two languages - "old Fortran" for maintain-

ing ancient programs, and "new Fortran" for program development. And I can understand their reasons; after all, if you're going to make wholesale changes, why not redesign the language from the bottom up (or from the top down) in order to achieve a tool which is consistent, clean, and well suited to what have been considered Fortran applications? But total redesign is repugnant to many people because it would be tantamount to admitting publicly that Fortran is dead. One can only wonder what would have happened if the early SHARE-IBM effort had retained the working name they applied to their product; what became PL/I was originally called Fortran VI.

Computer scientists in academic institutions, such as myself, are often characterized as walking around with our heads in the clouds, in contrast to those people who have to face the realities which "real users" confront. I do not believe that this view is justified. Most computer scientists earn their way through years of formal study by writing dozens of programs for applications from a variety of disciplines and in various languages, including Fortran. However, our academic training has also led us to understand that most questions about language use, definition and implementation have very few answers based on sound knowledge or theory. Therefore, we appreciate the necessity for exercising judgement in deciding among a variety of tools and techniques for implementing an application, because many of these possibilities offer very attractive tradeoffs with respect to the effectiveness of the final product.

As a result, I imagine that we secretly envy those for whom this task is eased by not having to determine which language is most appropriate for a given application, because that choice is often the most critical of all those which must be made in the light of our inadequate knowledge. I have the impression that the FOR-WORD thrust is to make that choice for all appropriate applications by changing Fortran in some significant ways, rather than making that choice on an individual basis for each application. You are demonstrating that language development is very much alive and that Fortran, as we have known it, is on the list of endangered species. I wish you very good luck in your efforts to reach an understanding in your quest for its successor.

Should "Structured Fortran" Imitate Algol?

B.A.M. Moon, Christchurch, New Zealand:

I found FOR-WORD No. 4 to be very interesting reading. It seems to me assured that the "ground-swell" generated by the interest and efforts of Fortran users will add significant new features to the Fortran language, of which the standard will take cognizance. Clearly the best of modern knowledge must go into the choice of those language features - for example, it is not sufficient merely to answer the criticism of Algol 60 advocates.

Algol 60 control structures also have their shortcomings, which are understandable as it is now an old language. Thus its loop structures allow greater flexibility than is justified by good programming standards and yet do not eliminate the need for the occasional GO TO. This is exemplified by the attached material from Professor Sale of the University of Tasmania:

"I shall stick my neck out further and detail some of the deficiencies of the for-loop. There are many of these. Some of the most obvious are:

"(1) There is no way in standard Algol 60 to write simple loops which have the escape test at the very loop start (see note (7) also), or at the very end of the loop, without resorting to gotos.

"(2) The ramifications of the *name* substitutions that pervade Algol 60 create mysterious and undesirable forms of loop that ought not to be written.

"(3) In the case of the step-until element, the elaboration of the Algol 60 report makes optimization of the loop a forbidding task; as a result most compilers make a very poor job of it. It is no easy task to cope with loops that may reverse counting directions in mid-flight, perhaps reselect count-variables, and perhaps move onto other list elements when this one is exhausted. It is foolish to pay such an exorbitant penalty for a feature we seldom want, and certainly never need.

"(4) The undefinition of the count variable after exhaustion of the particular element involved must now be regarded as a poor decision. The best approaches seem to be to fully define its value at all times (and forget the minor optimizations that are made more difficult), or even better, to make the count variable a fully local construct to the for-loop. This means any attempt to access it outside the loop is flagged by the compiler as a compile error; consequently programmers have to ensure that they save anything they wish to save in such cases. Even now, this lesson does not seem to have been fully learned: new languages still appear with loose undefined values proliferating themselves all over the place. This simply invites programmer errors.

"(5) The while element by itself creates a three component loop: the assignment, the test, and the loop body. These are executed in this order respectively.

"(6) Other problems can arise due to the non-obvious nature of this construct. For example, in:

```
FOR I:=1,I+1 WHILE I ≤ 10 DO S;
```

does the while test refer to both the 1 and I+1 type iterations, or just the I+1 iterations, or what?

"(7) In actual programming practice the strictures on the for-while construct make it less useful than it ought to be, since though S may be omitted (a dummy statement), the test cannot (except by writing the constant TRUE), and the assignment cannot be omitted nor expanded (though one can put rubbish or side-effect calls). Since in many loops the part of the loop preceding the escape test may be null, in standard Algol one has to resort to

```
FOR JUNK:=0 WHILE ABS(ERR) > TOLERANCE
   DO S;
```

or in other cases where a block of code is needed in that position, resort to goto constructs or even messier:

```
FOR JUNK:=FIRSTPARTOFLOOP WHILE ENDTEST
   DO S;
```

where FIRSTPARTOFLOOP is a procedure call containing the necessary statements. Ugh!

"I am afraid that the logical consequence to be drawn from all this is that rose-colored Algol blinkers are just as pernicious as Fortran dark glasses. All too often, the responses of programmers are conditioned by the languages they know and have used. There is a good case for breadth of experience here so that desirable changes in habits and languages are not inhibited by habit-formed ruts. The designers of Algol 60 did a good competent job for the early 60s, but to expect that work to stand as up-to-date for all time is surely ridiculous."

What is Fortran?

John Bolstad, Stanford CA:

Thanks for your Fortran newsletter No. 4. Your comment, "Apparently many people are not aware of the current status of standards for the Fortran language" is definitely an understatement. I'll bet not 1 percent of the programming population has any reasonable knowledge of the standards. I was, until recently, in this 99 percent. When I actually went to the trouble to look into the standard, I was appalled! For example, mixed mode arithmetic is not allowed to the right of the equal sign. I think most people would agree that this is a silly restriction, and in fact all of the compilers I have used (CDC, IBM, Xerox) allow it. I note in Engel's 1973 SIGPLAN Notices article that this is under consideration for the new standard.

Conversely, if one declares an array A(5, 5), it is perfectly permissible, according to the standard, to reference A(6, 1). I think this is rather unfortunate, especially since so many Fortran errors are caused by out-of-bounds memory references, but I think quite a few people would disagree with me on this.

You might be interested to know that the front part of the ACM Collected Algorithms contains reprints of the three articles about the Fortran standard that you mentioned were in Communications of the ACM.

In general, I think Fortran is deplorable, but, unfortunately, constraints such as portability and support force me to live with the "infantile disorder." At least MORTRAN masks a great deal of its ugliness.

Finally, I am greatly puzzled by references to the "spirit of Fortran." Do people also speak of the "spirit of the IBM 704" or "the spirit of 1954" or the "spirit of assembly language"? I think the problem is that most people have access to only one higher-level language, and this profoundly influences the way they think. I do not know anyone who has programmed in both Fortran and another high-level language (Cobol excepted) who does not feel ill when returning to Fortran. This is most evident here with beginning programming students, who are taught in Algol W. When they learn how it's done in the "real world", they are horrified. Therein, I think, lies our only hope for the betterment of programming languages.

Working Within Standard Fortran

Robert A. Ellis, St. Louis MO:

I was very interested to read in Datamation about the formation of a Working Group to address the issue of structured programming using Fortran. After observing that quite a bit can be done working strictly with standard Fortran (for example, if program modules are small, one listing page or less, very few GOTO's are needed and those that are used do not necessarily make the module difficult to follow), I would be most interested in being placed on your mailing list and would be interested in commenting on any proposals generated by your group.

Fortran Development as a Key to Structured Programming Principles

Waldo M. Wedel, Austin TX:

The SIGPLAN Fortran Development Committee is a long overdue effort in support of a widely used language. I see the greatest hope for helping large numbers of programmers understand many of the deeper principles of structured programming in this effort. Keep up the good work.

On Pre-compilers and Structures

H.D. Baecker, Calgary, Alberta:

I am not sure I understand the passion that is generated by proponents and opponents of pre-compilers. Most compilers have several passes, if you choose to call the first pass a pre-compiler that's OK by me. You can of course debate whether your first pass is efficient or not.

In the structured programming controversy I am not against GO TO, but for better constructs. I think that as an exit mechanism I find an explicit GO TO far easier to understand than some of the wierd constructs I have seen, as in Bliss. But that is just my personal taste.

The notion of a programming language that is in any manner wedded to a particular layout on cards or paper, where, say, particular card columns have special significance, is obscene. I can make enough errors without such unnecessary impositions. The total semantics of a program should be defined by marks on paper and its spatial properties should be of no significance whatever.

Agreed wholeheartedly that blocks (as opposed to procedures) should not have local variables. After implementing an Algol compiler, that is a pestilence I can do without, and I don't think it buys the user much. If your local block is that complicated it is probably clearer to write it as a procedure, and easier to debug thus.

In Favor of the Situation Case Statement

Redford Bond, Oklahoma City OK:

In my letter, excerpts from which were printed in FOR-WORD for August, I said that the n and a half repetition loop should not be implemented with Zahn's event indicators (now called a situation case statement) because it was unnatural, hard to learn and rigid.

After seeing this construct presented in a different way, in "Ill Chosen Use of 'Event'", it seems much more natural and easy to learn. I also find that it is quite flexible after reading some of Zahn's programming examples written in SGOL75.

Although the situation case statement will handle all loops, it is, as Knuth says, "big", and I still want for my novices the construct repeat S1 until B do S2 end, perhaps even generalized to repeat S1 until B1 do ... until Bn do Sn end.

If a concensus is reached that we need a situation case statement in structured Fortran, it then becomes necessary to decide how we will present the labels which are necessarily in such a construct. Statement numbers will not do; the advantage of the situation case statement is lost if the names of the situations do not convey meaning. Difficulties are posed by the fact that the use of the colon in Algol-like languages for labels may be foreign to our users; besides, it may not even be in the character set available to some of them. The following is suggested to provoke discussion and not as a recommendation:

```
until L1, L2, ... Ln do
    S0
then case
$L1$ S1
$L2$ S2
...
$Ln$ Sn
end
```

Vector preprocessor available

M. Wayne Wilson, Yorktown Heights NY:

I am enclosing a technical report on Vectran, co-authored by Dr. George Paul and myself. It is a superset of Fortran, designed within the philosophy of Fortran, to provide vector and array capabilities in a natural evolutionary manner, and with a minimum of new concepts. The report is available generally from: Mrs. Jan Douglass, Palo Alto Scientific Center, Palo Alto CA 94304.

A Subject for Debate?

Walter S. Brainerd, Pasadena CA:

I would like to see the following debated: "Fortran is ready for a major revision (not just adding more bells and whistles)." My first thought is that it is, but would like to hear some arguments. By the time the next standard is prepared, I hope we will have decided to some degree about such things as control structures. Perhaps the Development Group is an appropriate place to hold this debate.

An Efficient PL/I Compiler?

Lawrence D. Lopez, Urbana IL:

[From Introduction to "PLW Users Manual"] The PLW compiler translates the PLW language into FORTRAN for portability. The PLW language resembles PL/I in many respects. This was done in order to ease the transition for users who already know PL/I.

Procedures, DO loops, IF, and CASE statements are provided. List processing facilities and dynamic allocation of list elements are also provided although this facility is somewhat limited.

REVIEWS AND REFERENCES

Pratt, Terrence W. *Programming Languages: Design and Implementation* (Prentice-Hall, 1975)

This book is written from a very practical viewpoint, and it talks about issues in the implementation of real languages (including Fortran). The author believes that such questions as binding time and referencing environment are of crucial importance, while differences between the syntactic details of various languages are often mostly arbitrary. Therefore he spends little time discussing, for example, the various alternative methods of parsing arithmetic expressions.

About half of "Part I: Concepts" is devoted to three chapters on Data, Data Control, and Storage Management. Together, these give a very clear view of the implications of static and dynamic data structures upon the complexity and efficiency of the language processors that must cope with these structures. Especially lucid is the explanation of issues involved in passing information needed for the use of data structures between various program modules. It seems to be necessary to distinguish between various data control operations concerned with identifier associations, such as naming, activating, referencing, and accessing. All of these may vary according to the scope rules of the programming language.

Part II consists of separate chapters discussing the application of design and implementation concepts to Fortran, Algol-60, Cobol, PL/I, Lisp-1.5, Snobol-4, and APL. In these chapters, the issues are illuminated in terms of the vocabulary and of the general descriptions of Part I. One could perhaps debate the author's interpretation of some minor aspects of the ANSI Fortran standard specification. However, he gives a superb overall characterization of Fortran as simple, inelegant, weak in control structures, but efficient in execution; and he summarizes the history of the language to show how it came to have these properties.

The final chapter, on language universality, concludes that such desirable features as "clarity of structure, natural representations for problem data and algorithms, ease of extension, and efficiency" have to be compared in light of the purpose to which a language is applied.

"A carefully done treatise on programming languages"

Conway, R. and Gries, D. *An Introduction to Programming: A Structured Approach Using PL/I and PL/C* (Winthrop, 1973)

A splendid introductory programming text, far surpassing the majority in clarity of thought and direction. Contains a fairly simple description of loop invariants.

Ledgard, Henry F. *Programming Proverbs for Fortran Programmers* (Hayden, 1975)

This book has been reviewed in ACM Computing Reviews (see Nos. 28601 and 28602, Aug 75). Noted here will be some points that refer particularly to the Fortran language.

Proverb 6, "Prettyprint", points out that proper use of indentations for loops and other constructs, even in standard Fortran, can be an important aid to program comprehension. This point is expanded in chapter 4.

Any modern programming methodology book must of course include a discussion of top-down programming. Most writings to date, however, suffer from an overly simplistic approach which gives inadequate attention to the difficulties of this methodology, such as the need for backup and the interaction among decisions at the same level. As pointed out in the ACM review, this book suffers to some extent from this flaw.

The final chapter contains some useful comments on Fortran development. It includes a recommendation that Fortran programmers "use the GOTO, but use it sparingly," and a discussion that suggests we should conform to the standards we have, for the sake of portability, while we work toward incorporation of a number of badly needed features into an improved standard language. Features proposed here include "a simple quote convention for Hollerith strings, ... a good *if-then-else* structure, a facility for grouping compound statements, alphabetic statement labels, a facility for data structures, and numerous, now well-accepted control structures."

Einarsson, Bo *Aids for Software Generation and Evaluation -- A Draft Bibliography*

A nine-page bibliography by the chairman of IFIP Working Group 2.5. Includes Fortran, numerical software, software evaluation, preprocessors, etc. Available from the author at FOA Research Institute, Box 98, S-147-00 Tumba, Sweden.

Ryder, B.G. and Hall, A.D., *The PFORT Verifier: User's Guide* Bell Labs, Murray Hill, NJ [CS Tech Report #12]

The PFORT Verifier is a program which checks a Fortran program for adherence to PFORT, a portable subset of ANS Fortran. It diagnoses errors in inter-program-unit communication and Common usage which compilers often miss. The Verifier itself is written in PFORT and can easily be installed on a variety of computers. This paper describes the use of the Verifier and presents the portable subset in considerable detail.

Related information concerning syntax definition of Fortran is also available from Bell Labs.

= F O R - W O R D = >

Fortran Development Newsletter

Volume 1, Number 6; January 1976

Special Issue: PROPOSED ANS X3.9 FORTRAN LANGUAGE REVISION

Prepared for the U. S. Energy Research and
Development Administration under Contract W-7405-ENG-48

For additional copies of this report,
or for copies of other issues of
For-Word, please send inquiries to

Loren P. Meissner
50-B 3239
Lawrence Berkeley Laboratory
Berkeley CA 94720

TABLE OF CONTENTS

PROPOSED ANS X3.9 FORTRAN LANGUAGE REVISION

by

Loren P. Meissner
Lawrence Berkeley Laboratory
Berkeley, California 94720

1. INTRODUCTION

1.1. BACKGROUND

In 1966, after four years of effort, Fortran became the first programming language to be standardized in the United States. The 1966 Fortran standard was developed under the sponsorship of the American National Standards Institute, and is embodied in two documents known as American National Standard FORTRAN (ANS X3.9-1966) and American National Standard Basic Fortran (ANS X3.10-1966).

Work toward clarification of those documents was initiated in 1967, and it was later determined that efforts should be undertaken toward revision and extension of the 1966 standard.

By late 1975, the Fortran standards committee (ANSI X3J3) had produced a working document describing a new "full" Fortran language and a subset. This document has been forwarded to ANSI X3 as a draft proposed American National Standard FORTRAN, for further processing and possible replacement of the two 1966 documents. (The committee recommends that the single new document, consisting of a description of the full language and the subset, replace ANS X3.9-1966; and that ANS X3.10-1966 be withdrawn.)

This report is a paraphrase of the latter document. Please note that further review of the document, both by the committee and by the public at large, may result in further revision of the document before it is adopted as a new standard.

1.2. WHAT IS THE DOCUMENT?

1.2.1 Conformance [1.3]

The proposed revised Fortran standard document is primarily a specification of the form and interpretation of a "standard conforming" Fortran program. A standard conforming program in one which uses only language elements that are described in the document. A standard conforming processor, however, may also incorporate compatible extensions.

The document describes a full Fortran language and a subset ("subset Fortran"). With regard to the subset, a conforming program must be composed of language elements from the subest. A processor may contain extensions that are part of the full language, but only if they are implemented in a manner compatible with the full language. A subset processor may also include compatible extensions that are not part of the full language.

For example, section 3.3 states that "a statement must not contain more than 1320 characters." This means that if a programmer writes a longer statement, then his program is not standard conforming. This also implies that a standard conforming processor must accept statements up to 1320 characters long. It does not mean that a standard-conforming processor is prohibited from accepting longer statements. Accepting longer statements would be a compatible extension.

Prohibited features thus have the same status with respect to the standard as omitted features (i.e., those that are not mentioned at all in the document). For example, the document does not mention a double precision complex data type. Therefore, such a feature must not be used in a standard-conforming program. A standard conforming processor may or may not provide it or diagnose its use. The provision by a processor of such a feature would be a compatible extension.

1.2.2 Some features that are in the full language but not in the subset [22.3]

Double precision and complex types.

G, T, and -*n*X formats.

Alternate entry points and alternate returns for subprograms.

Generic intrinsic functions.

Parameter declarations -- i.e., compile time symbolic names for constants.

List-directed ("format free") input and output.

Format specification by reference to character variables and arrays.

Expressions in output lists.

Character substrings, concatenation operator, and functions of character type.

Variable length character items (analogous to adjustable dimensions) as dummy arguments.

BLOCK DATA subprograms.

Implied-DO lists in DATA statements.

DO parameters that are expressions or that are of real or double precision types.

(In many other contexts also, such as dimension declarators and subscripts, forms are restricted to integer constants and simple integer variables, and sometimes to a restricted class

of integer expressions.)

Note. In this report, the symbol "#" is used to mark items that are different in the subset.

1.2.3 Some restrictions of the subset language as compared to X3.9-1966

Only 9 continuation lines are permitted in a statement [3.3#].

DATA statements must precede all statement function statements (and all executable statements) [3.5#].

Double precision and complex types are not included [4.1#, 4.5#, 4.6#].

A format specifier must not be a reference to an array [12.4#].

The G edit descriptor is omitted [13.5#].

BLOCK DATA subprograms are not included. [16.1#, 16.2#].

2. THE FORTRAN LANGUAGE

2.1 LANGUAGE ELEMENTS

A Fortran program is composed of characters. Characters are grouped into lines; lines are grouped into program units; and program units are grouped into an executable program.

2.1.1 Fortran characters [3.1]

The Fortran character set consists of the 26 letters, 10 digits, and 13# special characters including blank and the following 12# symbols:

= + - * / ( ) , . ' $:

A partial collating sequence among the characters is specified: the letters are ordered from A to Z, the digits are ordered from 0 to 9, and the blank character is considered less than A and less than zero.

The blank character is not significant except (a) in a character constant or a character datum, (b) in an H or apostrophe edit descriptor in a FORMAT statement, and (c) in column 6 of a line to distinguish between an initial line and a continuation line of a statement.

Non-Fortran characters are permitted in certain contexts within a Fortran program, including columns 2 to 72 of a comment line, the characters of a character datum, and file names. However, the use of characters not in the Fortran character set may inhibit portability.

2.1.2 Lines [2.3, 3.2]

A line in a Fortran program consists of 72 character positions called columns. A line is a comment line, an initial line of a statement, or a continuation line of a statement.

A line with the letter C or an asterisk in column one is a comment line. A comment line may contain any Fortran characters or non-Fortran characters in columns 2 to 72.

A line that does not have the letter C or an asterisk in column one is an initial line if it has a zero or blank in column 6; otherwise it is a continuation line. Columns 1 to 5 of an initial line must contain a statement label or else be entirely blank. Columns 1 to 5 of a continuation line must be blank.

2.1.3 Statements [2.3, 3.3, 3.5]

A statement is written in columns 7 to 72 of one or more lines, the first of which must be an initial line, while the remaining ones, if any, must be continuation lines. Up to 19# continuation lines are permitted.

A comment line may precede a continuation line within a statement#.

An END statement must not have any continuation lines, and any other statement must not have an initial line that looks like an END statement.

A statement must not consist entirely of blanks. (However, it may have a blank initial line.)

A statement is executable or nonexecutable.

2.1.4 Executable statements [7.1]

Arithmetic, logical, and character assignment statements.

Statement label assignment (ASSIGN) statements.

Unconditional GO TO, assigned GO TO, and computed GO TO statements.

Arithmetic IF and logical IF statements.

CONTINUE statements.

STOP and PAUSE statements.

DO statements.

READ, WRITE, and PRINT statements.

REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE#, and INQUIRE# statements.

CALL and RETURN statements.

END statements.

2.1.5 Nonexecutable statements [7.2]

PROGRAM, FUNCTION, SUBROUTINE, ENTRY#, and BLOCK DATA# statements.

DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, PARAMETER#, EXTERNAL, INTRINSIC, and SAVE statements.

INTEGER, REAL, DOUBLE PRECISION#, COMPLEX#, LOGICAL, and CHARACTER type-statements.

DATA statements.

FORMAT statements.

Statement function statements.

Note. Although DIMENSION statements and type-statements are classified as nonexecutable, these statements may include expressions that are evaluated when the program unit is referenced.

2.1.6 Statement labels [3.4]

A statement may be identified by a statement label, which is a string of digits in a program. A statement label consists of one to five digits; leading zeros and imbedded or trailing blanks are ignored. Any statement may have a label, but the label of a nonexecutable statement except a FORMAT statement must not be referenced. Two statements in the same program unit must not have the same label.

2.1.7 Program units [2.4, 3.5]

A program unit consists of lines containing statements and comments, terminating with a final line which is an END statement.

A program unit is a main program or a subprogram. A subprogram is a function subprogram, a subroutine subprogram, or a block data# subprogram. An executable program is a collection of program units which contains exactly one main program.

2.18 Ordering of statements and lines [3.5]

Comment lines may appear anywhere# in a program unit, including ahead of the first statement.

A PROGRAM statement, if present, must be the first statement of a main program. The first statement of a subprogram must be a FUNCTION, SUBROUTINE, or BLOCK DATA# statement.

A FORMAT statement may appear anywhere, and an ENTRY# statement may appear anywhere in a function subprogram or subroutine subprogram, except within the range of a DO-loop.

All specification statements must precede all DATA statements, statement function statements, and executable statements.

All statement function statements must precede all executable statements.

DATA statements may appear anywhere after the specification statements#.

Among the specification statements, IMPLICIT statements must precede all other specification statements except PARAMETER# statements. A PARAMETER statement must precede all other statements containing the symbolic names# of constants that appear in that PARAMETER# statement.

The last line of a program unit must be an END statement.

2.2 STORAGE, DATA, AND CONSTANTS

2.2.1 Storage [2.13]

The concepts of storage unit and storage sequence, as used in the document, do not necessarily imply any particular realization or sequential arrangement of physical storage.

A storage unit is a character storage unit or a noncharacter storage unit.

2.2.2 Data types [4.1, 4.3 - 4.8]

Data is that which occupies storage. Each datum has a type, which is integer, real, double precision#, complex#, logical, or character.

An integer, real, or logical datum occupies one noncharacter storage unit. A double precision# or complex# datum occupies two consecutive noncharacter storage units.

A character datum occupies one or more consecutive character storage units. A character datum is a fixed-length string of character positions; the length of the string is the same as the number of character storage units occupied by the datum.

2.2.3 Variables, arrays, and substrings# [2.5 - 2.7, 5.1 - 5.3, 5.7]

A variable is a single datum.

An array is a sequence of data occupying consecutive storage units. Each datum in an array is an array element. All the elements of an array are of the same type. The position of a particular element within an array is designated by a subscript value.

A character datum may be a variable or an array element. The position of a particular character within a character datum is designated by a character position number.

A set of one or more consecutive character positions within a character datum is a substring#.

2.2.4 Definition status and value of a datum [2.11, 4.1, 4.3 - 4.8]

At any given time during execution of a program, each variable, array element, and substring# is either defined or undefined. A defined datum has a value, which does not change until the datum becomes undefined or is redefined. A datum of character type is defined if and only if all of its character positions are defined.

A datum may be initially defined by means of a DATA statement.

A standard-conforming program must not reference an undefined datum.

The value of an integer datum is an integer (whole number). The value of a real datum is a processor approximation to a real number. The value of a double precision# datum is a processor approximation to a real number which is more precise than that of a real datum.

The value of an integer, real, or double precision# datum may be positive, negative, or zero. The value zero is considered neither positive nor negative.

A complex datum# is an ordered pair of real data; the first element of the pair represents the real part and the second element represents the imaginary part.

The value of a logical datum is "true" or "false".

The value of each character position of a character datum is a representation of a Fortran or non-Fortran character. Blank characters are significant in a character datum.

2.2.5 Constants [4.2 - 4.8]

A constant is a string of digits and other characters in a program, representing a value that does not change. The form of a constant determines its value and its type.

An integer constant consists of an optional sign followed by a nonempty string of digits, to be interpreted as a decimal number.

A real constant is a basic real constant, a basic real constant followed by a real exponent, or an integer constant followed by a real exponent. A basic real constant consists of a string of digits containing a decimal point, with an optional sign. A real exponent is the letter E followed by an optionally signed integer constant, denoting a power of ten.

A double precision# constant is a basic real constant or an integer constant, followed by a double precision exponent. A double precision exponent is the letter D followed by an optionally signed integer constant, denoting a power of ten.

A complex# constant consists of an ordered pair of optionally signed real constants, separated by a comma and enclosed in parentheses.

A logical constant has the form ".TRUE." or ".FALSE."

A character constant consists of a nonempty string of characters enclosed between apostrophes. (The enclosing apostrophes are not included in the string.) An apostrophe within the string is represented by a pair of adjacent apostrophes.

2.3 NAMES

2.3.1 Symbolic names [2.1, 2.6, 5.1, 8.6, 18.1, 18.2]

A symbolic name in a program consists of from one to six alphanumeric characters, the first of which must be a letter. Note that file names, and some other sequences of characters such as format edit descriptors and keywords, are not symbolic names even though they may have the same form as a symbolic name.

A symbolic name is the name of a constant#, a variable, an array, a common block, a program unit, or a procedure.

A symbolic name of a constant# is specified in a PARAMETER# statement. An array name is declared in an array declarator. A common block name is declared in a COMMON statement.

A program unit name is a main program name, a subroutine subprogram name, a function subprogram name, a subprogram entry# name, or a block data# subprogram name. A main program name appears in a PROGRAM statement. A subroutine subprogram name is declared in a SUBROUTINE statement. A function subprogram name is declared in a FUNCTION statement. A subprogram entry# name is declared in an ENTRY statement in a subroutine or function subprogram. A block data# subprogram name is declared in a BLOCK DATA# statement.

A procedure name is a subroutine procedure name, an external function procedure name, an

intrinsic function name, or a statement function name. The name of a subroutine subprogram, a function subprogram, or a subprogram entry is a procedure name when it is used in certain contexts, namely: in an executable statement as a reference or as an actual argument, in an EXTERNAL or INTRINSIC statement, in a type-statement, or as a dummy argument.

A subroutine procedure name or an external function procedure name will also appear in the same executable program as a program unit name, unless it is the name of a subprogram specified by means other than a Fortran subprogram.

A subroutine procedure name is used in a CALL statement to reference a subroutine subprogram or subprogram entry#. An external function procedure name is used in an expression to reference a function subprogram or function subprogram entry#.

An intrinsic function name is used in an expression to reference a specific or generic# intrinsic function. A name in the "Specific name" or "Generic name"# column of the table of intrinsic functions is classified as an intrinsic function name, except in a program unit in which the name appears in a conflicting type statement or in an EXTERNAL statement.

A statement function name appears on the left side of a statement function statement, and may also be used in an expression in the same program unit to reference the statement function.

A symbolic name that is not the name of a constant#, an array, a common block, a program unit, or a procedure is a variable name. A name that appears in a function subprogram as a program unit name is classified as a variable name wnen it appears elsewhere in the same program unit.

Note that a symbolic name that appears in a program unit only within a dummy argument list and in an actual argument list may be the name of a variable, an array, a subroutine procedure, an external function procedure, or an intrinsic function. If a name appears only in an EXTERNAL statement and in an actual argument list, it may be a subroutine procedure name or an external function procedure name.

2.3.2 Data type of a name [4.1 - 4.8, 5.2, 8.6, 15.2 - 15.5, 15.7, 15.10]

The name of a constant#, a variable, an array, a function subprogram, a function subprogram entry#, an external function procedure, a statement function, or an intrinsic function (except a generic# intrinsic function) has a definite data type in a program unit. The name of a common block, a subroutine subprogram, a subroutine subprogram entry#, a main program, a block data# subprogram, or a subroutine procedure, does not have a data type.

The type of the symbolic name of a constant# is determined by the form of the constant rather than by the form of the symbolic name.

The type of the name of a variable, an array, a function subprogram, a function subprogram entry#, an external function procedure, or a statement function may be established explicitly in a type-statement. The type of the name of a function subprogram may also be established explicitly by the FUNCTION statement. For a name whose type is not established in one of the foregoing ways, the type is established implicitly, and depends only upon the first letter of the symbolic name. The implicit type of a name is established by default or by an IMPLICIT statement. The default implicit type is integer if the first letter is I, J, K, L, M, or N; otherwise the default implicit type is real. An entity of character type has a length, which is specified explicitly or implicitly along with the type.

The type of a function procedure name determines the type of the value supplied by the procedure reference in an expression. (A procedure name that appears as a dummy argument of a function subprogram, a subroutine subprogram, or a subprogram entry# must not be of character type.)

The type of a specific intrinsic function name is given in the table of intrinsic functions [15.10]. The name may also appear in a non-conflicting type-statement; its appearance in a conflicting type-statement removes it from the class of intrinsic function names for that program unit. The type of a generic# function name depends upon the type of the argument (as shown in the table), and may vary within a program unit.

2.3.3 Association [17.1]

Association provides more than one name for a given datum.

Two variables, array elements, or substrings# are associated if they share at least one character or noncharacter storage unit. Two entities are totally associated if they both occupy exactly the same storage units. Two entities are partially# associated if some, but not all, of their storage units are shared.

An EQUIVALENCE statement causes association of entities in a program unit. A COMMON statement causes association of entities in different program units. Procedure references cause association between dummy and actual arguments. In an external function subprogram, there is an association between the function subprogram name,

all entry# names, and all local variable names that are the same as the function subprogram name or an entry# name.

Associated entities need not have the same type; however, a character entity must not be associated with a noncharacter entity.

Partial association# of noncharacter entities may exist between a double precision# or complex# entity on one hand and an integer, real, logical, double precision#, or complex# entity on the other hand. Such association must not occur through argument association, but may be specified by an EQUIVALENCE, COMMON, or ENTRY# statement.

Partial association# of character entities may occur through argument association, or may be specified by an EQUIVALENCE, COMMON, or ENTRY# statement.

2.3.4 Scope of names [2.9, 18.1, 18.2]

The scope of a program unit name, or of a common block name, is an executable program. An entity is a global entity if the scope of its name is an executable program. Two global entities in the same executable program must not have the same symbolic name.

The scope of the symbolic name of a constant#, a variable, an array, or a procedure is a program unit, with the following exceptions: The scope of the name of a dummy argument of a statement function is the statement function statement. The scope of an implied-DO# variable in a DATA statement is the implied-DO# list in which it appears. An entity is a local entity if the scope of its name is a program unit.

Two local entities in a program must not have the same symbolic name. However, a local entity in one program unit may have the same name as a local entity in another program unit; and a local entity in a program unit may have the same name as a global entity whose name does not appear in that program unit.

A local entity in a program unit must not have the same name as a global entity whose name appears in that program unit, with the following exceptions:

(1) A function subprogram name or entry# name in a function subprogram may be the same as the name of a local variable in the same subprogram.

(2) A common block name in a program unit may be the same as an array name, a subroutine procedure name, a statement function name, or an external function name in the same program unit. A common block name in a program unit may be the same as a variable name in that program unit, except for a variable name that is the same as the subprogram name or an entry# name. When a common block name in a program unit is the same as the name of a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity.

An entity may have a name whose scope is less than a program unit. The scope of the name of a dummy argument of a statement function is the statement function, and the scope of the name of an implied-DO# variable in a DATA statement is the implied-DO list.

Two or more entities may have the same name if the scope of each is less than a program unit provided that no part of the program unit is included in the scope of the name of more than one of the entities. Such an entity may also have the same name as a local entity or a common block in the same program unit.

An IMPLICIT statement or a type-statement in a program unit, including the length specification for a character entity, also applies to names whose scope is less than the program unit. Use of a name that appears in the table of intrinsic functions, as a name whose scope is less than a program unit, removes it from the class of intrinsic function procedure names for that program unit.

2.4 ARRAYS AND SUBSTRINGS

2.4.1 Array names [2.6]

The symbolic name of an array refers to a sequence of consecutive data called array elements. Each array element may be referenced by the array name qualified by a subscript, whose value specifies the particular element within the array. All the elements of an array are of the same type, which is the type of the array name.

2.4.2 Array declarator [5.1]

An array declarator is used to designate a symbolic name as an array name. It consists of the array name followed by a parenthesized list containing at least one and not more than seven# dimension declarators. Each dimension declarator consists of an upper dimension bound expression, optionally preceded by a lower# dimension bound expression and a colon#. The number of dimensions of the array is the number of dimension declarators in the array declarator.

Each dimension bound expression must be composed of integer constants, symbolic names# of integer constants, and integer variable names. (No function references, array element names, or exponentiation operations are permitted.) Integer variable names are permitted only in adjustable array declarators. If a lower# dimension bound is omitted, the default lower

bound is one. The value of each upper dimension bound must be greater than that of the corresponding lower bound.

An array name must not appear in more than one array declarator in a program unit. An array declarator may appear in a DIMENSION statement or a type-statement. If the array name is not a dummy argument in a function or subroutine subprogram, the array declarator may appear in a COMMON statement.

2.4.3 Array element names [5.3]

An array element name consists of an array name followed by a subscript, which is a parenthesized list of subscript expressions (separated by commas), one for each dimension of the array.

Each subscript expression is an integer, real#, or double precision# expression, and may contain array element# references or function# references. The value of a real# or double precision expression# is truncated to determine the effective subscript value. This value must not be less than the corresponding lower dimension bound, nor greater than the corresponding upper dimension bound.

2.4.4 Array element ordering [5.2, 5.4]

The elements of an array are ordered according to the sequence of their subscript values. The subscript value is determined from the subscript expression values along with the dimension bounds, according to the formula

$$V = 1 + \Sigma_{i=1}^{n} (s_i - l_i) \Pi_{j=1}^{i-1} (u_j - l_j + 1)$$

where s_i is the (integer part# of the) ith subscript expression, l_i is the ith lower dimension bound, and n is the number of dimensions. (Note that if each s_i equals l_i then V is one.) If each s_i equals u_i, then the value of V is equal to the number of elements in the array.

2.4.5 Use of array names [5.6]

An array name may be used (without a subscript) to identify the entire array in certain contexts. These are: (1) in an input or output list; (2) as a dummy argument; (3) as an actual argument in a subroutine or external function procedure reference; (4) in a COMMON statement; (5) in a type-statement; (6) as the format identifier in an input or output statement#; (7) in an EQUIVALENCE statement; (8) in a DATA statement; (9) in an array declarator; (10) in a SAVE statement; (11) as the unit identifier for an internal file in a READ or WRITE statement#.

2.4.6 Character substrings# [5.7]

The character positions in a string (i.e., in a variable or array element of character type) are consecutively numbered, in storage sequence order, beginning with position 1. (The positions of characters in a character constant are numbered from left to right.) A substring# is a set of characters having consecutively numbered positions within a string.

A substring name consists of a character variable name or a character array element name, followed by a parenthesized substring specification. There must be a colon inside the parenthenes; to the left and right are optional substring expressions specifying the smallest and largest character position numbers, respectively, for the substring. If there is no expression to the left of the colon, the default smallest character position is one; if there is no expression to the right of the colon, the default largest character position number is the length of the string.

The substring specification expressions, if present, must be of integer, real, or double precision type. The value of a real or double precision expression is truncated to determine the effective substring expression value. The smallest character position number must be at least one, and must be less than or equal to the largest character position number, which in turn must not exceed the length of the string.

2.5 SPECIFICATION STATEMENTS AND DATA STATEMENTS

2.5.1 DIMENSION statement [8.1]

A DIMENSION statement consists of the keyword DIMENSION followed by a list of array declators separated by commas. Array declarators may also appear in COMMON statements or type-statements.

2.5.2 EQUIVALENCE statement [8.2]

An EQUIVALENCE statement specifies association of entities within a program unit. It consists of the keyword EQUIVALENCE followed by a list of equivalence classes separated by commas.

Each equivalence class is enclosed in parentheses, and consists of two or more variable names, array element names, array names, or character substring# names, separated by commas. (Note that the 1966 Fortran standard allowed the name of a multiply dimensioned array to appear with a single subscript in an equivalence class.) When a program unit name is also used as a variable name in a function subprogram, that name must not appear in an equivalence class.

Character and noncharacter entities must not appear in the same equivalence class. Character entities of different lengths may appear in the same class.

Appearance of names in an equivalence class may cause association of other entities whose names do not appear explicitly, due to storage sequence rules for array elements, characters in a string, items in a common block, etc. An EQUIVALENCE statement must not contradict other storage sequence specifications implied by such rules.

2.5.3 COMMON statement [8.3]

A COMMON statement specifies association of entities in different program units, and may also contain dimension information. It consists of the keyword COMMON followed by a list of common classes.

Each common class consists of a common block name followed by a list of variable names, array names, and array declarators, separated by commas. The variable names and array names must not be subprogram dummy arguments.

Each common block name appears between a pair of slashes. A pair of slashes with nothing between them may also be used to specify the blank common block; and if the first common class in a COMMON statement refers to the blank common block, the first pair of slashes may be omitted.

If a common block name (or omitted name specifying blank common) appears more than once in the COMMON statements of a program unit, the lists of names in all the common classes for that common block name are interpreted as forming a single common class in the order of their appearance.

Character and noncharacter entites must not be included in the same common class.

A common block storage sequence is formed from all of the storage units occupied by the variables and arrays whose names appear in a common class. This storage sequence may be extended beyond the last storage unit, if necessary, to include storage units occupied by entities associated by equivalence association. Such associated entities must not occupy storage units that precede the first storage unit of a common block sequence.

Within an executable program, all common blocks with the same common block name (but not the blank common block) are required to have a storage sequence that includes the same number of storage units.

Entities in named common blocks may be initially defined by means of DATA statements, but entities in blank common must not be initially defined. Execution of a RETURN or END statement in a subprogram may cause entities in named common blocks, but not in blank common, to become undefined.

An equivalence class in an EQUIVALENCE statement must not include entities from more than one common block.

2.5.4 Type-statements [8.4]

A type-statement explicitly establishes the type of the name of a variable, an array, a function subprogram, a function subprogram entry#, an external function procedure, or a statement function. A character type statement also specifies length.

More than one explicit type specification for a name within a program unit is prohibited. When an explicit specification for a function subprogram name appears in the FUNCTION statement, any other explicit type specification for the name within that program unit is prohibited.

The type of the symbolic name# of a constant is determined by the form of the constant. Explicit type specifications for such names are prohibited.

If a conflicting type specification is given for a specific intrinsic function name, then the name is not an intrinsic function name in the program unit. If any explicit type specification is given for a generic# intrinsic function name, then the name is not a generic# function name in the program unit.

A non-character type statement consists of the keyword INTEGER, REAL, DOUBLE PRECISION#, COMPLEX#, or LOGICAL, followed by a list of variable names, array names, array declarators, function subprogram names, function subprogram entry# names, statement function names, or function procedure names (separated by commas).

A character type-statement consists of the keyword CHARACTER, followed by an optional length specifier, followed by a list of character variable names, character array names, character array declarators, character function# subprogram names, character function# subprogram entry# names, or character function# procedure names (separated by commas). Each name or declarator in the list is followed by an optional length specifier.

If a length specifier immediately follows the keyword, it applies to each name or declarator in the list that does not have its own length specifier. (This first length specifier, if present, may be followed by an optional comma.) Any entry for which no length is specified has the length one by default. Note that all elements of a character array have the same length.

A length specifier consists of an asterisk followed by one of the following: an unsigned, nonzero integer constant; an integer expression enclosed in parentheses; or an asterisk in parentheses#. If an integer expression is used, it must be composed of integer constants, symbolic names of integer constants, and integer variable names#. (No function references or array element references are permitted.) An expression# containing a variable name is permitted only in an adjustable length specifier#.

The length specifier in a character type statement for a function# subprogram name or function entry# name may be an adjustable length specifier, or it may contain an asterisk in parentheses#. In the latter case, the function assumes the length specified in the referencing program unit. In any case, the length specified in the referencing program unit must not conflict with the specification in the subprogram.

A character entity whose length specifier is an adjustable length# specifier or an asterisk enclosed in parentheses# must not be used as an operand of a concatenation operator# except in a character assignment statement.

2.5.5 IMPLICIT statement [8.5]

An IMPLICIT statement specifies rules for establishing the implicit type of the name of a variable, an array, a function subprogram, a function entry#, an external function procedure, or a statement function, based upon the first letter of the name. These rules also establish implicit lengths for character entities. Implicit type rules do not apply to intrinsic functions, nor to symbolic names# of constants.

An IMPLICIT statement consists of the keyword IMPLICIT, followed by a list of implicit type specifications separated by commas. Each implicit type specification consists of the keyword INTEGER, REAL, DOUBLE PRECISION#, COMPLEX#, LOGICAL, or CHARACTER, followed by a parenthesized list of single letters or ranges of letters, separated by commas.

An IMPLICIT statement specifying CHARACTER type may include an asterisk and an integer constant expression# denoting the implicit length, between the keyword CHARACTER and the parenthesized list; if the implicit length specification is omitted, the default implicit length is taken as one.

A range of letters is denoted by a pair of letters (which must be in alphabetical order) separated by a minus, and is interpreted as including all letters between those mentioned.

A letter must not appear more than once, as a single letter or within a range of letters, in all the IMPLICIT statements of a program unit.

2.5.6 PARAMETER statement# [8.6]

A PARAMETER statement specifies the symbolic name of a constant. It consists of the keyword PARAMETER followed by a list of parameter specifications, separated by commas. Each parameter specification consists of a symbolic name, an equals sign, and a constant, in that order. The symbolic name of a constant must not appear to the right of the equals sign. A name must not appear more than once as a symbolic name in the PARAMETER statements of a program unit.

The type of the symbolic name of a constant is determined by the form of the constant. The length specification for the symbolic name of a character constant is the length of the character constant.

2.5.7 EXTERNAL statement [8.7]

An EXTERNAL statement permits an external procedure name to be used as an actual argument, or it indicates the existence of an external procedure or block data# subprogram having the same name as an intrinsic function.

An EXTERNAL statement consists of the keyword EXTERNAL, followed by a list of external procedure names or block data# subprogram names, separated by commas. (Note that the name of a statement function is not permitted.) A symbolic name must not appear more than once in the EXTERNAL statements of a program unit.

2.5.8 INTRINSIC statement [8.8]

An INTRINSIC statement permits an intrinsic function name to be used as an actual argument. An INTRINSIC statement consists of the keyword INTRINSIC, followed by a list of intrinsic function names separated by commas. The names of intrinsic functions in the "MAX" or "MIN" families must not be used as actual arguments (because the functions in those families have an indefinite number of arguments). The appearance of a generic# function name in an INTRINSIC statement does not remove the generic property. A function must not appear more than once in the INTRINSIC statements and EXTERNAL statements of a program unit.

2.5.9 SAVE statement [8.9]

A SAVE statement maintains the definition status of entities in a subprogram upon execution of a RETURN or END statement. A SAVE statement must not appear in a main program or in a BLOCK DATA# subprogram. A subprogram may contain more than one SAVE statement.

Entities whose definition status is maintained may include common blocks, variables, and arrays. Variables and arrays within a common block must not be included except by specifying the entire block.

A SAVE statement may consist simply of the keyword SAVE. This specifies that the definition status of all common blocks, variables, and arrays in the program unit (but not that of dummy arguments) is to be maintained.

The list form# of the SAVE statement consists of the keyword SAVE followed by a list of common block names (enclosed between slashes), variable names, and array names, separated by commas. A name must not appear more than once in the SAVE statements of a program unit. The names of dummy arguments, of procedures, or of variables or arrays within a common block, must not appear in a SAVE statement.

2.5.10 DATA statement [9.1 - 9.4]

A DATA statement provides initial values for variables, arrays, array elements, and substrings#. At the beginning of execution of an executable program, all entities in all program units are undefined except those for which initial values have been provided in DATA statements.

A DATA statement consists of the keyword DATA followed by one or more pairs (optionally separated by commas), each pair consisting of a list of names followed by slashes enclosing a list of constants.

A list of names includes variable names, array names, array element names, substring# names, and implied-DO lists#, separated by commas. The list must not contain names of dummy arguments, functions, or entities in blank common (or entities associated with those in blank common). Also prohibited is the name of a variable in an external function subprogram that is also the function name or an entry# name (or is associated with one of these).

A list of constants consists of items separated by commas; each item is a constant or the symbolic name# of a constant, or one of these preceded by a repeat count and an asterisk. A repeat count is a nonzero, unsigned integer constant or the symbolic name# of such a constant.

An implied-DO list# in a DATA statement consists of a list of array element names and implied-DO lists, followed by a control part, all enclosed in parentheses. The control part includes an implied-DO variable which must be an integer variable, an equals sign, and two or three expressions separated by commas. The iteration count is established from these expressions exactly as for a DO-loop, except that the iteration count must not be zero. An implied-DO-variable in a DATA statement is local to the implied-DO list. Each of the expressions, as well as each subscript expression in the list part of the implied-DO, is an integer constant expression, except that it may contain implied-DO variables of enclosing implied-DO lists.

There must be a one-to-one correspondence between the names in the list of names and the constants in the list of constants, in each pair in a DATA statement. In this correspondence, an array name in the list of names includes all elements of the array, an implied-DO list# includes the names of all referenced array elements, and repeat counts are applied to items in the list of constants.

The type of each name must agree with the type of the corresponding constant when either is of type character, complex#, or logical. A constant of integer, real, or double precision# type may correspond with a name of any of these types; type conversion# will be applied to the constant if necessary. A character constant will be padded with blanks on the right or truncated# from the right, if necessary, to the length of the corresponding character entity in the list of names.

An entity, or two associated entities, must not be initially defined more than once in an executable program.

2.6 EXPRESSIONS AND ASSIGNMENT

2.6.1 Arithmetic expressions [6.1]

An arithmetic primary is an unsigned arithmetic constant, a symbolic name# of an arithmetic constant, a variable reference, an array element reference, a function procedure reference, or an arithmetic expression enclosed in parentheses.

A factor consists of one or more primaries separated by the exponentiation operator "**". In the interpretation of a factor containing two or more exponentiation operators, the primaries are combined from right to left.

A term consists of one or more factors separated by either the multiplication operator "*" or the division operator "/". In the interpretation of a term containing two or more multiplication or division operators, the factors are

combined from left to right.

An arithmetic expression consists of one or more terms separated by either the addition operator "+" or the subtraction operator "-". The first term may optionally be preceded by the identity (unary "+") or the negation (unary "-") operator. In the interpretation of an expression containing two or more addition or subtraction operators, the terms are combined from left to right.

Among the arithmetic operators, the exponentiation operator has highest precedence, the multiplication and division operators have intermediate precedence, and the addition and subtraction operators have lowest precedence.

The data type of an expression containing one or more arithmetic operators is determined from the data types of the operands. In an expression consisting of an operand preceded by an identity or negation operator, the type of the expression is the same as the type of the operand. For an expression consisting of a pair of operands separated by an addition, subtraction, multiplication, or division operator, the type of the expression is determined as follows: If both operands are of the same type, then the type of the expression is the same also. If either operand is of integer type, then the type of the expression is the same as the type of the other operand. If one operand is of real type and the other is of double precision# or complex# type, then the type of the expression is double precision# or complex# respectively. An operator must not have one operand of double precision type and the other operand of complex type.

The type of an expression consisting of a pair of primaries separated by an exponentiation operator is determined by the same rules. However, a complex operand is prohibited in an exponentiation operation except for the case of a complex operand raised to an integer power.

Except for a value raised to an integer power, type conversion is applied to the operand (if any) that differs in type from the resulting expression. No type conversion is required when a value of any type is raised to an integer power.

Note that the type of an expression consisting of an operator operating on a single operand or on a pair of operands is determined by the types of those operands -- i.e., "locally" -- and not by the type of any operand in any larger expression containing it.

If the quotient of two expressions of integer type is not a whole number, it is truncated to the next lower integer in magnitude.

2.6.2 Character expressions [6.2]

A character primary is a character constant, a symbolic name of a character constant#, a character variable reference, a character array element reference, a character substring# reference, a character function procedure reference#, or a character expression# enclosed in parentheses.

A character expression consists of one or more primaries separated by the concatenation# operator "//". In the interpretation of a character expression containing two or more concatenation# operators, the primaries are combined from left to right. The operands for a concatenation operator must have constant length except in a character assignment statement.

Note that parentheses have no effect in a character expression.

2.6.3 Relational expressions [6.3]

A relational expression consists of a pair of arithmetic expressions or a pair of character expressions, separated by one of six relational operators: .LT. (less than), .LE. (less than or equal to), .EQ. (equal to), .NE. (not equal to), .GT. (greater than), or .GE. (greater than or equal to).

A relational expression involving a pair of arithmetic expressions of different types is interpreted as comparing the difference of the two expressions with zero. Complex expressions# may be compared only with the .EQ. and .NE. relational operators. (A **complex** expression must not be compared with a double precision expression.)

A relational expression involving a pair of character expressions of different lengths is interpreted as if the shorter operand were extended on the right with blanks to the length of the longer operand.

2.6.4 Logical expressions [6.4]

A logical primary is a logical constant, a symbolic name# of a logical constant, a logical variable reference, a logical array element reference, a logical function procedure reference, a relational expression, or a logical expression enclosed in parentheses.

A logical factor is a logical primary, or the logical negation operator ".NOT." followed by a logical primary.

A logical term is a sequence of logical factors separated by the logical conjunction operator ".AND." In the interpretation of a logical term containing two or more logical conjunction operators, the logical factors are combined from left to right.

A logical expression is a sequence of logical terms separated by the logical disjunction operator ".OR." In the interpretation of a logical expression containing two or more logical disjunction operators, the logical terms are combined from left to right.

Among the logical operators, the logical negation operator has highest precedence, the logical conjunction operator has intermediate precedence, and the logical disjunction operator has lowest precedence.

2.6.5 Precedence of operators [6.5]

Precedence within the class of arithmetic operators, and within the class of logical operators, is established by the interpretation rules for expressions. There is only one character operator#. A relational expression contains exactly one operator. In an expression containing operators of more than one of these classes, arithmetic operators have highest precedence, followed by character operators, then by relational operators, and finally by logical operators which have the lowest precedence.

2.6.6 Evaluation of expressions [6.6]

Any datum referenced as an operand in an expression must be defined at the time the reference is executed, and the type of the datum must agree with the type of the name used to reference it. An integer operand must have an integer value (i.e., it must not have a statement label value as a result of the execution of an ASSIGN statement). Any arithmetic operation whose result is not mathematically defined is prohibited in a standard-conforming program.

Side-effects of functions must not alter the value of any other entity within the same statement, and must not alter the value of any entity in a common storage area that affects the value of any other function reference in the same statement. In particular, if an actual argument is defined during execution of a function, that argument or any associated entities must not appear elsewhere in the statement.

(An exception is made, however, in that function references in the logical expression of a logical IF statement may have side-effects that affect the contingent statement.)

A processor is required to evaluate only as much of an expression as is necessary to determine the value of the expression. For character expressions#, a processor needs to evaluate only as many characters of the result as are required by the context.

If a function reference appears in a part of an expression that does not need to be evaluated, all entities that would become defined during execution of the function become undefined when evaluation of the expression is completed.

If a statement contains more than one function reference, the references may be executed in any order, except for the ordering specified for expressions in input or output list elements, logical IF statements, and function argument lists containing function references. The value provided by each function reference must be independent of the order of evaluation of the references.

2.6.7 Equivalent expressions [6.7]

A processor may evaluate an expression different in certain respects from that obtained by application of the interpretation rules. However, any expression contained in parentheses must be treated as an entity.

For arithmetic expressions, the processor may evaluate any expression that is mathematically equivalent to that obtained from the interpretation rules, provided that the integrity of parentheses is respected. Integer division is not considered mathematically equivalent to real division. [Note that there is no mention in the document of the equivalence or non-equivalence of real and double precision expressions.]

For relational expressions, the processor may evaluate any expression that is relationally equivalent.

For logical expressions, the processor may evaluate any expression that is logically equivalent, provided that the integrity of parentheses is respected.

2.6.8 Arithmetic, logical, and character assignment statements [10.1, 10.2, 10.4]

An arithmetic assignment statement consists of the name of a variable or an array element of integer, real, double precision#, or complex# type, followed by an assignment operator "=", followed by an arithmetic expression.

A logical assignment statement consists of the name of a variable or an array element of logical type, followed by an assignment operator "=", followed by a logical expression.

A character assignment statement consists of the name of a variable, an array element, or a substring# of character type, followed by an assignment operator "=". followed by a character expression. Character positions within the datum designated on the left of the assignment operator must not be referenced by the expression on the right.

Execution of an assignment statement be-

gins with evaluation of the expression to the right of the assignment operator. The rules for evaluation of expressions require that this expression has a defined value. The datum designated by the variable name, array element name, or substring name# to the left of the assignment operator becomes defined with this value, after conversion if necessary, upon execution of the assignment statement.

In the case of arithmetic assignment, conversion may consist of type conversion, whose effect is the same as applying the appropriate generic type conversion function. For character assignment, the value of the expression will be padded with blanks on the right or truncated from the right, as necessary to achieve the required length.

Execution of a character assignment statement having a substring# on the left defines only the characters of the specified substring#. The definition status of other characters of the same string is not changed by the execution of the assignment statement.

2.6.9 ASSIGN statement [10.3]

An ASSIGN statement consists of the keyword ASSIGN, a statement label, the keyword TO, and an integer variable name (in that order). The statement label must be the label of an executable statement or a FORMAT statement in the same program unit.

Execution of an ASSIGN statement defines the integer variable, with the statement label as its value. The variable becomes undefined with respect to use in integer (arithmetic) expressions or in any other way, except for reference by an assigned GO TO statement or as a format identifier in an input or output statement. The variable may later be redefined with another statement label value, or with an integer (arithmetic) value.

2.6.10 Events that cause entities to become defined [17.2]

Execution of an arithmetic, logical, or character assignment causes the entity named on the left of the assignment operator to become defined.

As execution of an input statement proceeds, each entity that is assigned a value (of the correct type) from the input medium becomes defined at the time of this assignment.

Execution of a DO statement causes the DO-variable to become defined.

Beginning execution of an implied-DO list in an input or output statement causes the implied-DO variable to become defined.

A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.

Execution of an INQUIRE# statement causes any entity that is assigned a value during execution of the statement to become defined if no error condition occurs.

Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

When an entity of a given type becomes defined, all totally associated entities of the same type become defined.

A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined and if the actual argument has a form compatible with the dummy argument.

When a complex# entity becomes defined, all partially# associated real entities become defined. If both parts of a complex# entity become defined as a result of partially associated real or complex entities becoming defined, the complex# entity becomes defined.

2.6.11 Events that cause entities to become undefined [17.3]

All entities are undefined at the beginning of execution of an executable program, except those entities initially defined by DATA statements.

When an entity of a given type becomes defined, all totally associated entites of different types become undefined.

Execution of an ASSIGN statement causes the variable in the statement, as well as any associated entities, to become undefined for use as integers.

When an entity of type other than character becomes defined, all partially# associated entities become undefined, except for partial association between a complex# entity and a real or complex# entity.

If a reference to a function appears in an expression in which the value of the function is not needed to determine the value of the expression, and if evaluation of the function would cause an argument of the function of an entity in common to become defined, then the argument or the entity in common becomes undefined when the expression is evaluated.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become unde-

fined, except for the following: (a) entities in blank common; (b) initially defined entities that have neither been redefined nor become undefined; (c) entities specified by SAVE statements; (d) entities in named common blocks that appear in the subprogram and in at least one other program unit that is referencing the subprogram either directly or indirectly. Note that any integer variable to which a statement label value has been assigned becomes undefined upon execution of a RETURN or END statement. Note that at the beginning of execution of an external function subprogram, all local variables whose names are the same as the function name or an entry# name are undefined.

When an error condition or an end-of-file condition occurs during execution of an input statement, all of the entities in the input list of the statement become undefined.

Execution of a direct access input statement that specifies a record that has not been previously written causes all of the entities in the input list of the statement to become undefined.

Execution of an INQUIRE# statement may cause entities to become undefined.

When an entity becomes undefined as a result of the foregoing conditions, all totally associated entities, and all partially associated entities of types other than character, become undefined.

2.7 CONTROL STATEMENTS

2.7.1 The execution sequence [3.6]

Execution of a program begins with the first executable statement appearing in the main program, and continues with the statements in order of their appearance except when the sequence is interrupted by execution of a procedure reference or a control statement.

A subprogram procedure reference causes interruption of execution of the program unit where the reference appears; execution continues from the first executable statement following the subprogram header statement or ENTRY# statement referenced. A statement function reference causes execution of the statement function statement.

A control statement is a GO TO, arithmetic IF, RETURN, or STOP statement, an input or output statement containing an error specifier# or an end-of-file specifier (statement label), a CALL statement with an alternate return specifier#, a logical IF statement containing any of the foregoing, a DO statement, the terminal statement of a DO-loop, or an END statement.

2.7.2 GO TO statements [11.1 - 11.3]

An unconditional GO TO statement consists of the keyword GO TO followed by the statement label of an executable statement in the same program unit. Execution of a GO TO statement causes interruption of the sequence of execution of statements; execution continues with the statement having the designated label.

A computed GO TO statement consists of the keyword GO TO, a parenthesized list of statement labels (separated by commas), an optional comma, and an integer, real#, or double precision# expression# in that order. Execution of a computed GO TO statement begins with evaluation of the expression, and conversion# (if necessary) to integer type. If the (integer) value of the expression is between 1 and the number of labels in the list, then the effect is an unconditional GO TO, to the label whose position in the list corresponds to the value of the expression. Otherwise, the execution sequence continues with the statement following the computed GO TO.

An assigned GO TO statement consists of the keyword GO TO, followed by an integer variable name. (This may optionally be followed by a parenthesized list of labels of statements in the same program unit, separated by commas; an optional comma may appear between the variable name and the left parenthesis.) At the time of execution of an assigned GO TO, the integer variable must be in a defined state (as a result of the execution of an ASSIGN statement) and its value must be the statement label of an executable statement in the same program unit. If the parenthesized list of labels is present, the value of the integer variable must be one of the labels in the list. Execution of an assigned GO TO statement has the effect of an unconditional GO TO to the designated label.

2.7.3 IF statements [11.4 - 11.5]

An arithmetic IF statement consists of the keyword IF, followed by a parenthesized integer, real, or double precision# expression, followed by the labels of three executable statements in the same program unit (separated by commas). The expression is evaluated, and the execution sequence continues with the statement having the first, second, or third label, according as the value of the expression is less than, equal to, or greater than zero, respectively.

A logical IF statement consists of the keyword IF, followed by a parenthesized logical expression, followed by a contingent statement. The contingent statement must be an executable statement, but it must not be a DO statement, a logical IF statement, or an END statement. The logical expression is evaluated, and the contin-

gent statement is executed if the logical expression is true. If the logical expression is false, the contingent statement is ignored. (Note that the logical expression may contain references to functions having side effects; such side effects are permitted to affect the value of the contingent statement.)

2.7.4 DO statement and DO-loops [11.6]

A DO statement is the opening statement of a DO-loop. It consists of the keyword DO, followed by a statement label and an optional comma, followed by a control part. The control part includes a DO-variable, an equals sign, and two or three expressions of integer, real#, or double precision# type separated by commas.

The statement label references the terminal statement of the DO-loop, which must appear following the DO statement in the program unit, and which must be an executable statement but must not be an unconditional GO TO statement, an assigned GO TO statement, an arithmetic IF statement, a RETURN statement, a STOP statement, an END statement, or a DO statement.

The DO-variable is an integer, real#, or double precision# variable, and each of the control expressions is an integer, real#, or double precision# expression#.

The range of a DO-loop consists of all executable statements in the program unit that are in the block following the DO statement and ending with the terminal statement. If a DO statement appears within the range of a DO-loop, the corresponding terminal statement must also be within the same range. A statement may be the terminal statement of more than one DO-loop.

A DO-loop is either active or inactive. It is initially inactive, and becomes active only during execution of its DO statement. An active DO-loop becomes inactive only when one of the following occurs:

(1) its iteration count is found to be zero during loop control processing;

(2) its DO-variable becomes undefined or is redefined by means other than incrementation processing;

(3) its DO statement is in the range of another DO-loop that becomes inactive;

(4) its DO statement is in the range of another DO-loop whose DO statement is executed; or

(5) a RETURN, STOP, or END statement in the program unit is executed.

When a DO-loop becomes inactive, its DO-variable retains its value (if any).

Note that transfer of control to statements outside the range does not cause a DO-loop to become inactive (unless the DO-variable is redefined or becomes undefined).

Transfer of control into the range of an inactive DO-loop is prohibited. Transfer to a statement in the range of one or more DO-loops all of which are active is permitted. (Note that if a statement is in the range of two or more DO-loops, some of which are active while others are inactive, the ranges of all of these inactive loops must be contained in the ranges of all of the active loops. The inner loops cannot be reactivated without execution of statements in the range of some active outer loop.)

Execution of a DO statement includes all of the following steps:

(1) The expressions in the control part are evaluated and converted#, if necessary, to the type of the control variable. The (converted#) values of the first two expressions become the values of m_1, the intial parameter, and m_2, the terminal parameter, respectively. If there is a third expression, then its value (which must not be zero) becomes the value of the incrementation parameter, m_3; otherwise the default value of m_3 is one.

(2) The DO-variable becomes defined, with the value of m_1. The DO-loop becomes active, and all DO-loops whose DO statements are contained within its range become inactive.

(3) The initial value of the iteration count is established, as the value of the expression

$$max\ (int\ ((m_2 - m_1 + m_3) / m_3),\ 0)$$

(4) Loop control processing, as described in the following paragraph, completes the execution of the DO statement.

Loop control processing consists of testing the iteration count. If it is greater than zero, then execution of the range of the DO-loop begins. If the iteration count is zero, the DO-loop becomes inactive. If, as a result, all of the DO-loops sharing the terminal statement of this DO-loop are now inactive, normal execution continues with execution of the next executable statement following the terminal statement. However, if some of the DO-loops sharing the terminal statement are active, execution continues with incrementation processing, as described below.

Execution of the range of a DO-loop begins at the first executable statement following the DO statement in the program unit. When (if ever) the terminal statement is reached, it is executed. Unless execution of the terminal statement results in a transfer of control, execution then proceeds with incrementation processing, as described in the following paragraph.

Incrementation processing includes the following steps:

(1) Among the active DO-loops sharing this terminal statement, that one is selected whose DO statement was executed most recently. (The foregoing description implies that the loop selected will be the innermost active loop sharing this terminal statement.)

(2) The value of the DO-variable of the selected loop is incremented by the value of the incrementation parameter m_3.

(3) The iteration count for the selected loop is decremented by one.

(4) Execution continues with loop control processing (described above) for the selected DO-loop.

Note. A loop with m_1 less than m_2 was prohibited in the 1966 standard, but most processors extended the language to permit such loops, and interpreted them so that the statements in the range of the loop were executed once. The effect of this widely implemented extension can be acheived with a program that conforms to the new language description, by writing

DO *label* *index* = m_1, MAX (m_1, m_2)

2.7.5 Other control statements [11.7 - 11.10]

CALL and RETURN statements are described later (see "MAIN PROGRAM AND SUBPROGRAMS").

A CONTINUE statement consists of the keyword CONTINUE. Execution of a CONTINUE statement has no effect.

A STOP statement consists of the keyword STOP, optionally followed by a string of not more than five digits, or by a character constant. Execution of a STOP statement causes termination of execution of the executable program, and makes the string of digits or the character constant (if any) accessible.

A PAUSE statement consists of the keyword PAUSE, optionally followed by a string of not more than five digits, or by a character constant. Execution of a PAUSE statement causes interruption of the execution of the executable program, in such a way that execution can be resumed, and makes the string of digits or the character constant (if any) accessible.

An END statement consists of the keyword END, written only in columns 7 to 72 of an initial line. An END statement must not have a continuation line. The last line of every program unit must be an END statement. If an END statement is executed, its effect is that of a RETURN statement in a subprogram, or a STOP statement in a main program.

3. INPUT AND OUTPUT

3.1 CONCEPTS

3.1.1 Files [12.2]

A file is unformatted or formatted. Data transferred to or from an unformatted file is not edited. Data transferred to or from a formatted file is edited, either by an explicit format (specified by a reference in a READ, WRITE, or PRINT statement to a FORMAT statement or to a character datum containing a format specification) or by list-directed# formatting (specified by an asterisk used as a format specifier in the READ, WRITE, or PRINT statement.)

A file is an internal file, a sequential access (external) file, a stream access# (external) file, or a direct access (external) file. An internal file is formatted, and is edited by an explicit format. A sequential access file is formatted or unformatted; if formatted, it is edited by an explicit format or by list-directed# formatting. A stream access# file is formatted, and is edited by list-directed formatting. A direct access file is formatted# or unformatted; if formatted#, it is edited by an explicit format.

3.1.2 Records [12.1]

A record of an internal file is a character variable, character array element, or character substring#. If the file has more than one record#, the file must be a character array consisting of one record for each element of the array. Each input or output statement begins with the first record of the file; if the statement transfers more than one record#, these are read or written in array element sequence.

The records of a sequential access file or of a stream access file# are read and written in record number order.

Each record of a direct access file has a unique record number, which is established when the record is written and does not change. Records may be written in any order. Each input or output statement transfers data to or from one record. There is no way to delete a record, but a record may be rewritten. A file has a maximum record number, which is specified either explicitly or by default.

The length of a record of a sequential access or stream access# file is established at the time the record is written.

The length of a record of an internal file is the length of the character datum containing the record. If the number of characters written is less than the length of the character datum, the remainder is filled with blanks.

The length of a record of a direct access file is a property of the file, and is the same for all records of the file. If the values specified by the output list do not fill the record, the remainder is filled with blanks if the file is formatted#, or with integer zeros if the file is unformatted.

Writing a record to an internal file causes the corresponding character datum to become defined. A character datum that is a record of an internal file may also become defined or be referenced by means other than input or output statements. A record must not be read if the corresponding character datum is undefined.

On a formatted sequential access file, records written with explicit format editing must not be read by list-directed format editing#, and records written with list-directed format editing# must not be read. A BACKSPACE statement must not be executed on a file if the previous record was written with list-directed# formatting.

3.1.3 Unit specifier [12.3]

A unit specifier has the form [UNIT =]# unit, where unit is a reference to an internal file or to an external unit.

A reference to an internal file is the name of the character variable, character array#, character array element, or character substring# containing the data of the internal file.

A reference to an external unit is an integer, real#, or double precision# expression# which, when truncated (if necessary) to an integer has a positive or zero value which is associated with a particular external unit. The integer value associated with an external unit is the same throughout an executable program, and is used for all references to a file connected to the unit.

3.1.4 File position [12.9, 12.10]

A direct access file is implicitly positioned during execution of a READ, WRITE, or PRINT statement. Prior to data transfer, the file is positioned at the initial point of the record specified in the control list. This record becomes the current record during data transfer. After data transfer, if no error has occurred, the file is positioned at the end of this record, and the nextrec specifier in an INQUIRE# statement may be used to determine the record number of the record following the one just written.

An internal file is implicitly positioned during execution of a READ, WRITE, or PRINT statement. Prior to data transfer, the file is

positioned at the initial point of its first record. This record becomes the current record when data transfer begins. If more than one# record is transferred, these are read or written in array element sequence. After data transfer, if no error has occurred, the file is positioned just beyond the last record read or written [however, this fact is not useful].

A stream access file# is implicitly positioned during execution of a READ, WRITE, or ENDFILE statement, and may be explicitly positioned by execution of a REWIND statement. Execution of a READ or WRITE statement changes the position of the file during data transfer, but not prior to data transfer or after data transfer. If no error has occurred, the position after data transfer is just after the last value read or written. A stream access file# consists of zero or more formatted records, followed by at most one endfile record. Execution of an ENDFILE statement on a stream access file# creates an endfile record as the next record of the file, and positions the file after the endfile record. Execution of a REWIND statement positions the file at its initial point.

A sequential access (external) file is implicitly positioned during execution of a READ, WRITE, PRINT, or ENDFILE statement, and may be explicitly positioned by execution of a BACKSPACE or REWIND statement. The position of the file is not changed prior to data transfer. Each data transfer statement begins a new record. The file will have been positioned just ahead of some record, which becomes the current record during data transfer. (On output, the current record is a previously empty record that becomes the last record of the file.) One or more records are read or written during formatted data transfer; one record is read or written during unformatted data transfer. If no error occurs, the position after data transfer is just after the last record read or written and that record becomes the preceding record. (If the last value transferred by a READ statement is within a record, the file is moved to the end of that record following data transfer.) Execution of an ENDFILE statement on a sequential access (external) file creates an endfile record, and positions the file after the endfile record. Execution of a REWIND statement positions the file at its initial point. Execution of a BACKSPACE statement causes the file to be positioned before the preceding record (if any); if there is no preceding record, the position of the file is unchanged.

3.1.5 REWIND, BACKSPACE, and ENDFILE [12.10]

A control list for a REWIND, BACKSPACE, or ENDFILE statement has either of the following forms:

unit

(clist)#

where clist# consists of a unit specifier, or of a unit specifier and an error specifier. (An error specifier has the form ERR = label, where label is the label of an executable statement in the same program unit. See Error Detection, below.)

A REWIND, BACKSPACE, or ENDFILE statement consists of the appropriate keyword followed by a control list of either of the foregoing forms.

A BACKSPACE statement must reference a unit that is connected to a sequential access file that exists.

Execution of a REWIND statement for a file that is connected but does not exist, is permitted but has no effect.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

3.1.6 Error detection [12.6]

An error specifier has the form ERR = label, where label is the label of an executable statement in the same program unit. When an error is detected during execution of an input or output statement containing an error specifier, control is transferred to the statement having the specified label.

An error is detected in the following cases: (The processor may also detect other conditions.)

An attempt to execute a READ or WRITE statement referencing a direct access file, with a record number less than one or greater than the maximum record number for the file.

An attempt to execute an unformatted WRITE on a formatted direct access file, or a formatted WRITE on an unformatted direct access file.

An attempt to transfer more data to or from a record than the record holds.

An attempt to execute an OPEN statement with improperly valued specifiers.

Note. The position of a file, upon detection of an error, becomes indeterminate.

3.1.7 Restrictions on input and output statements [12.11 - 12.12]

An input or output statement must not contain a function reference that causes a further input or output statement to be executed.

A function reference anywhere in a list item or specifier in an input or output statement must not have any side effects that affect the value of any other entity in the same statement.

An input or output statement must not reference a unit if the unit or the file connected to the unit does not have all of the properties required for execution of the statement.

3.2 READ, WRITE, AND PRINT# STATEMENTS

3.2.1 Statement forms [12.8]

A READ, WRITE, or PRINT# statement has one of the following forms:

READ (clist) [list]

READ fmt [, list]#

WRITE (clist) [list]

PRINT fmt [, list]#

A control list, clist, has one of the following forms:

unit, fmt, kwlist

unit, kwlist

kwlist

where unit and fmt are unit and format specifiers in non-keyword form, and kwlist is a list of specifiers all of which are in keyword form. The control list must include a unit specifier, and may include at most one of each of the following: a format specifier, an error specifier, and either an end-of-file specifier or a record specifier. A format specifier is included for a formatted file. A record specifier is included for a direct access file. An end-of-file specifier may be included in a READ statement for a sequential access or stream access# file.

The forms that do not include control lists must not be used for a direct access file (since a record number specification is needed, and can only be given in a control list).

The optional list must be present if list-directed editing# is specified, or for a WRITE statement to an unformatted file.

3.2.2 Specifiers [12.3 - 12.7]

A specifier in a READ, WRITE, or PRINT# statement is a unit specifier, a format specifier, a record specifier, an error specifier, or an end-of-file specifier. Unit specifiers and error specifiers are described above.

A format specifier (for a formatted file) has the form [FMT =]# fmt, where fmt is one of the following:

(1) the statement label of a FORMAT statement in the same program unit;

(2) the name of an integer variable# that has a statement label value referencing a FORMAT statement in the same program unit;

(3) a character constant;

(4) a character array name#,

(5) a character expression# that does not involve concatenation of variable-length operands; or

(6) an asterisk.

A record specifier (for a direct access file) has the form REC = rec, where rec is an integer, real# or double precision# expression# whose integer part is positive.

An end-of-file specifier (for a READ statement on a sequential access or stream access# file) has the form EOF = eof, where eof is the label of an executable statement in the same program unit.

3.2.3 The input or output list [12.8]

The list of a READ, WRITE, or PRINT# statement specifies the data whose values are to be transferred.

A basic list item in a READ statement is a variable name, an array element name, a character substring# name, an array name, or an array block item#. A basic list item in a WRITE or PRINT statement is any of the foregoing, or any expression# except a character expression that involves concatenation of variable length operands.

An array block item# includes a colon. To the left and right of the colon, respectively, are specifications of the first and last array element in the block. Each of these specifications is an array element name, except that one (but not both) may be omitted. If both are present, they must be names of elements of the same array, and the subscript value for the element name on the right must be greater than for the

one on the left. If the specification on the left is omitted, the default first array element of the block is the first element of the array; if the specification on the right is omitted, the default last element of the block is the last element of the array.

An array name or an array block item# appearing as a basic list item is equivalent to a sequence of basic list items that includes in order all the array elements of the array or of the array block#.

Each item of an input or output list is a basic list item or an implied-DO list. An implied-DO list consists of a list of basic list items or implied-DO lists, followed by a control part, all of which is enclosed in parentheses. The control part includes an implied-DO-variable, an equals sign, and two or three expressions separated by commas. The implied-DO-variable is an integer, real#, or double precision# variable, and the control expressions are integer, real# or double precision# expressions. The iteration count and the sequence of values of the implied-DO-variable are established from these expressions exactly as for a DO-loop. In an input list, the implied-DO-variable must not appear as a list item in the same implied-DO list. The implied-DO list is equivalent to a sequence of items that includes the list items once for each iteration of the implied-DO list.

3.2.4 Execution [12.9]

Data transfer is specified by the input or output list and, in some cases, by the format. File positioning may occur even when the list is omitted and no data transfer is specified by the format. A READ statement transfers data from an internal file, or from a unit connected to an external file. A WRITE or PRINT statement transfers data to an internal file, or to a unit connected to an external file.

A READ or PRINT statement that does not contain a control list is interpreted as if it included a default unit specifier for an external unit or for an internal file. The default unit specifier is processor-determined. A default file for the PRINT statement must be a sequential external file.

For a READ statement that includes a unit specifier, the specifier must refer either to an internal file, or to an external unit that is connected to a file at the time the statement is executed.

A READ, WRITE, or PRINT statement for a formatted file includes a format specifier. If it is a reference to an explicit format, the referenced FORMAT statement or character datum controls editing during data transfer. If the format specifier is a reference to a character datum and the unit specifier (or default specifier) refers to an internal file, the datum containing the format specification must not be contained within the internal file.

If the format specifier is an asterisk, it specifies list-directed editing#; therefore the file for the specified (or default) unit must be a sequential access external file or a stream access# file, and the input or output list must not be omitted.

All values needed to determine which entities are specified by a list item are established at the beginning of the processing of that item. All data specified by a list item is transferred prior to the processing of any succeeding list item. A DO-variable of an implied-DO list becomes defined at the beginning of processing of the items of that implied-DO list.

All data referenced in an output list must be defined. An attempt to read a record of a direct access file that has not previously been written causes all entities specified by the input list to become undefined.

If the format specifier is a reference to a character datum, an input list item or any associated entity must not include any part of the character datum that contains a part of the format specification.

If the unit specifier (or default unit specifier) refers to an internal file, an input or output list item or any associated entity must not be contained within the internal file.

Unformatted data transfer causes exactly one record to be read or written. No editing occurs during data transfer.

One or more records are written during formatted data transfer. For a direct access file, the record number is increased by one as each succeeding record is read or written.

A WRITE or PRINT statement may transfer formatted data to a sequential (external) file connected to a unit (or default unit) that is a printer. The first character of each record is not printed, but is used to determine vertical spacing. The remaining characters, if any, are printed on one line (beginning at the left margin). Vertical spacing before printing is one line (normal) if the first character is blank, two lines if the first character is "0" (zero), skip to first line of next page if the first character is "1" (one), and no advance if the first character is "+" (plus). A record with no characters has the same effect as a record containing a single blank character.

3.3 EXPLICIT FORMATTING

3.3.1 Format specification [13.1 - 13.2]

An explicit format is specified by a reference to a format statement or to a character datum containing a format specification. A reference to a format statement may consist of the label of the FORMAT statement appearing in the READ, WRITE, or PRINT statement, or of a reference in the statement to an integer variable# that has a statement label value referencing the FORMAT statement. A reference to a character datum is either a character constant in the READ, WRITE, or PRINT statement, or a character array name#or character expression# in the statement, specifying a character string that is a valid format specification.

A FORMAT statement must be labelled, and consists of the keyword FORMAT followed by a format specification enclosed in parentheses.

If a format is specified by a reference to a character datum, the datum must be defined with a string consisting of matching left and right parentheses that enclose a valid format specification. Blank characters may precede the left parenthesis, and character positions following the right parenthesis may contain arbitrary characters or may be undefined. If the reference is to a character variable# or character array element#, the format specification must be contained within the referenced datum. If the reference is to a character array#, the format must be contained within the array, but may continue beyond the first array element into other consecutive elements.

The format specification consists of a list of items, each of which is either an edit descriptor or a format specification enclosed in parentheses, optionally preceded by a repeat count specification (which is a nonzero, unsigned integer constant). The items in the list are separated by commas, except that the comma may be omitted between a P edit descriptor and an immediately following F, E, D# or G# edit descriptor, before or after a slash edit descriptor, or before or after a colon edit descriptor. Each edit descriptor is one of the following:

I *w*	# D *w.d*	# T *c*
# I *w.m*	# G *w.d*	*b* X
F *w.d*	L *w*	# S, SP, SS
E *w.d*	A	*k* P
# E *w.d* E *e*	A *w*	BN, BZ
# E *w.d* D *e*	/	# :
'h_1 h_2 ... h_n'	*n* H h_1 h_2 ... h_n	

where h_1, h_2, ..., h_n are Fortran or non-Fortran characters; and *w*, *m*# *d*, *e*# *c*# *b*, and *k* are integer constants. (All of these constants except *m*# *d*, and *k* must be nonzero; all except *b* and *k* are unsigned, and these two are optionally signed.) The T# X, S# P, BN, BZ, H, slash, colon# and apostrophe edit descriptors must not be preceded by a repeat count specification.

3.3.2 Interaction with input or output list [13.3]

Explicitly formatted input or output involves a sequence of input or output list items, and a sequence of edit descriptors. If the READ, WRITE, or PRINT statement that references the format specification contains an input or output list, then the format specification must include at least one I, F, E, D# G# L, or A edit descriptor.

Each input or output list item comprises one element of the list sequence (taking into account all of the items specified by array names, array block items, and implied-DO lists), except that each list item that is a reference to a datum or expression of complex type comprises a pair of consecutive elements of the list sequence.

Each format specification item that is preceded by a repeat count *r* has the same effect as if the item were written *r* times consecutively in the format specification.

The sequence of edit descriptors is as written, from left to right, taking into account all repeat specifications. In effect, the last edit descriptor on the right is followed by an implicit colon and slash.

Each I, F, E, D# G# L, or A edit descriptor (counting repetitions) is associated with one element of the input or output list sequence. The T# X, S# P, BN, BZ, H, slash, colon# and apostrophe edit descriptors are not associated with any element of the list sequence.

If the number of associated edit descriptors is insufficient, then a "rescan point" is determined as follows: If any item of the format specification is a format specification enclosed in parentheses, then the rescan point is at the beginning of the last such item in the main list, and includes its repeat count (if any). Othewise, the rescan point is the beginning of the format specification. The total effective format specification is the original format specification, followed by as many repetitions as are required of that portion of the original specification to the right of the rescan point. Each repetition includes the implicit colon and slash at the end. (If this repetition is required, the repeated part must contain at least one I, F, E, D# G# L, or A.)

Format control begins with the first edit descriptor in the effective format specification, and continues in sequence.

When a T# X, S# P, BN, BZ, H, slash, or apostrophe edit descriptor is encountered, the appropriate editing is performed without reference to the input or output list sequence.

When an I, F, E, D# G# L, A, or colon descriptor is encountered, reference is made to the input or output list sequence. If the sequence has been completed, format control terminates. If there are items remaining, then if the edit descriptor is a colon# nothing happens; otherwise, the associated item from the input or output list sequence is edited as required.

3.3.4 Interaction with a file [13.5]

Each record of a formatted file is a sequence of characters. A field is a part of a formatted record that is transmitted to or from the file as a result of a single edit operation. A field corresponds to one I, F, E, D#, G# L, or A edit descriptor and to one item in the input list sequence, or to one H or apostrophe edit descriptor (with no corresponding list item). The field width is the number of characters in the field.

Numeric input editing. Leading blanks in a field are not significant. Plus signs are optional. A field of all blanks is interpreted as zero.

An I w or I w.m#edit descriptor is associated with an input list item of integer type. The value of w specifies the field width; m# is ignored.

An F w.d, E w.d, E w.d E e# E w.d D e# or G w.d#edit descriptor is associated with a real or double precision#item in the input list sequence. The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point, optionally followed by an exponent consisting of an E or D#followed by an optionally signed integer constant. If no decimal point appears in the input field, the value of d is used to determine the implied position of the decimal point, counting from the last character position of the field, or the last character preceding the exponent if there is one in the field. If there is a decimal point in the field, it overrides the d specification of the edit descriptor. The input field may contain more digits than are considered significant by the processor.

The P edit descriptor specifies a scale factor, which is an optionally signed integer constant. The value of the scale factor in effect at the beginning of execution of an input or output statement is zero. The effective scale factor value does not change during execution of the statement, except when a P edit descriptor is encountered. The scale factor has no effect upon input, except for F, E, D# or G#editing of a field that does not contain an explicit exponent. If there is no exponent in the input field, the field is interpreted as if it were followed by the exponent -k.

Numeric output editing. A (nonzero) negative value produces a minus sign in the output field; a zero value does not produce a minus sign; a positive or zero value has an optional plus sign. The external representation is right justified in the field, with leading blanks. If the number of characters exceeds the field width, or if an exponent exceeds its specified length using E w.d E e#or E w.d D e# editing, then the entire output field is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, then the field is not filled with asterisks. (Note that a plus sign is not an optional character when SP sign control# is in effect.)

An I w or I w.m#edit descriptor is associated with an input or output list sequence item of integer type. The value of w specifies the field width. The value of m# if present, must not exceed that of w; this value specifies the minimum number of non-blank characters to be transmitted, including leading zeros if necessary. If m#is zero, a zero datum will be transmitted as a field of blank characters regardless of the sign control#in effect.

An F w.d edit descriptor is associated with an output list sequence item of real or double precision type. The field width is specified as w, with a fractional part consisting of d digits. The output field consists of blanks, if necessary, followed by a minus sign or an optional plus sign, followed by a string of digits that includes a decimal point and represents the magnitude of the data value, as modified by the scale factor currently in effect, and with the fractional part rounded to d decimal digits.

An E w.d, D w.d, E w.d E e# or E w.d D e# edit descriptor is associated with an output list sequence item of real or double precision# type. The field width is w, with a fractional part consisting of d digits, and an exponent of e#digits. The output field consists of an optional sign, followed by an optional zero, followed by a decimal point, followed by d significant digits obtained by rounding the datum, followed by an exponent.

The exponent may have any of several forms. If e#is not specified, the exponent value must

not exceed 999; if it exceeds 99, the exponent occupies four character positions including a sign and three digits. If e# is specified, the exponent occupies e# + 2 character positions, including a D# or E (whichever letter precedes the e# specifier), a sign, and e# exponent digits. If e# is not specified and the exponent is 99 or less, the exponent occupies four character positions; either it is in the same form as if the exponent exceeds 99 (with an extra zero leading exponent digit) or it is in the same form as if e# were specified with the value 2 (and with an E in the output field for E w.d editing, and with either an E or a D# for D w.d# editing).

The scale factor is specified by a P edit descriptor, in the same manner as for numeric input editing.

For F output editing, the scale factor value is added to the exponent of the internal value.

For E or D# output editing, the scale factor must be greater than -d and less than d + 2. If $-d < k \leq 0$, there will be exactly $|k|$ leading zeros and $d - |k|$ significant digits after the decimal point. If $0 < k < d + 2$, there will be exactly k significant digits to the right of the decimal point and $d - k + 1$ significant digits to the right of the decimal point (thus a total of d + 1 significant digits in the output field).

A G w.d# edit descriptor is associated with an output list sequence item of real or double precision# type. The field width is specified as w, with a fractional part consisting of d digits. Editing is the same as F or the same as E, depending upon the magnitude of the datum. If the exponent would be between 0 and d, inclusive, the F mode is used, but the value of w is decreased by 4 and the value of d is adjusted so that the total number of significant digits, both to the right and to the left of the decimal point, is the original value of d in the G# edit descriptor; the scale factor has no effect in this case. If the exponent is negative or is larger than d, then editing (including scale factor effects) is identical to E w.d .

L editing. An L w edit descriptor is associated with a list item of logical type. The value of w specifies the field width. The input field consists of one or more blanks, followed by a T for true or an F for false. Characters in the field after the first non-blank character are not significant. The output field consists of w - 1 blanks followed by a T for true or an F for false.

A editing. An A or A w edit descriptor is associated with a list sequence item of character type. The value of w, if present, specifies the field width; the default field width is the length of the character datum specified as the list item. If w is specified, the string will be truncated from the right or padded with blanks on the right, if necessary, according to the length of the character datum.

Apostrophe editing. Apostrophe editing must not be used on input. The field width is the same as the number of characters in the edit descriptor when interpreted as a character constant. The specified character string is transmitted to the output record, without reference to the output list.

H editing. This form of editing must not be used on input. The n H edit descriptor causes the n characters (including blanks) immediately following the H in the format specification to be transmitted to the output record. The field width is n.

T# and X editing. These edit descriptors specify the position, within the record, of the next character transmitted. The position may be in either direction with respect to the current position. On input, the specified position may be beyond the end of the record if no characters are transmitted from those positions. On output, the effect is as if the record were initially filled with blank characters, which are replaced as specified during editing. On output, an X edit descriptor causes all skipped positions to be transmitted, and therefore may increase the length of the record. A T# edit descriptor does not change the length of the record.

The T c# edit descriptor indicates that the character position of the next character to be transmitted to or from the record will be c .

The b X edit descriptor indicates that the character position is to be increased or decreased# by b positions relative to the current position.

Slash editing. The slash edit descriptor causes the input or output file to be positioned at the initial point of the next record, which then becomes the current record.

Colon editing# The colon# edit descriptor terminates format control if there are no more items in the input or output list sequence. If there are more items, this edit descriptor has no effect.

S editing#. The S# edit descriptor has no effect on input. For output, it affects only I, F, E, D# and G# editing. The output fields produced by these edit descriptors include positions in which the processor may insert an optional plus sign. At the beginning of execution of each output statement, the processor option is in effect. An SP (sign print) edit descrip-

tor# requires the processor to insert plus signs in those positions; an SS (sign suppress) edit descriptor# requires that those positions be left blank; and an S (sign optional) descriptor restores the option to the processor.

BN and BZ editing. These specifiers have no effect on output. On input, they affect only I, F, E, D# and G#editing. Blank characters, other than leading blanks, in fields edited by these descriptors may be interpreted as zeros or ignored. At the beginning of execution of each input statement, the interpretation is as specified when the file was connected (unless changed by execution of a subsequent OPEN). A "BN" edit descriptor causes blanks to be ignored (except that a field of all blanks has the value zero); a "BZ" edit descriptor causes blanks to be interpreted as zeros.

3.4 LIST-DIRECTED FORMATTING

List-directed formatting is specified by a format specifier consisting of an asterisk.

3.4.1 List-directed input [13.6]

The characters in one or more records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant.

Each value is a constant, a null value, the form r * c, or the form r * , where c is a constant representing a value and r is an unsigned, nonzero integer constant representing a repeat count. These forms must not contain embedded blanks, except where permitted within a constant.

A value separator is a (maximum-length) nonempty string of characters consisting of zero or more blanks and at most one comma or slash.

Each constant must be of a form acceptable for the corresponding input list item; an integer constant is also acceptable corresponding to a real or double-precision list item.

A constant corresponding to a complex list item must consist of a pair of real constants or integer constants enclosed in parentheses and separated by a comma. Blanks are permitted only within a character constant, or on either side of the comma separating the two parts of a complex constant. The end of a record, when it is not within a character constant, has the same effect as a blank; when it is within a character constant, it is ignored, except that that it must not separate a pair of consecutive apostrophes denoting a single apostrophe. A character constant may be of different length than the corresponding list item; the character constant will be truncated from the right or padded with blanks on the right if necessary.

A null value is specified by successive value separators with no constant between them or with the form r * . The list item corresponding to a null value retains its previous value (or remains undefined). A single null value may represent an entire complex constant, but must not be used for the real or imaginary part separately. Initial blanks read at the beginning of execution of a list-directed input statement are not significant, and are not interpreted as constituting an initial null value followed by a separator.

When a slash is encountered within a separator during list-directed input, execution of the statement terminates. The effect is as if null values had been supplied for any remaining list items.

3.4.2 List-directed output [13.6]

Values are produced in sequence, as specified by the output list. Except for character constants, values are separated by a string of characters consisting of blanks and at most one comma (as specified by the processor). The record structure of the output file is determined by the processor. The processor may begin a new record at any point except within a constant, and under certain conditions may begin a new record within a character constant or complex constant.

For a list item of logical type, the character T or F is produced.

For a list item of integer type, a string of decimal digits is produced.

For a list item of real or double precision type, a constant is produced in approximately the same form as by G w.d editing, for some processor-defined values of w and d.

For a list item of complex type, a pair of real constants is produced, enclosed in parentheses and separated by a comma. If the entire constant is not longer than an entire record, blanks or the end of a record must not occur within a complex constant. If the constant is longer than a record, blanks may be inserted between the comma and the end of the record, and one blank may appear at the beginning of the following record.

The form of a character constant is different for a sequential file and for a stream file.

List-directed output to a sequential file is "printable": character constants are not delimited by apostrophes, are not preceded or followed by value separators, have each apostrophe in the string represented by one apostrophe in the file, and have blank characters (for vertical spacing control) inserted at the beginning of any record that contains the continuation of a character constant from a preceding record; character constants in this form are not suitable for list-directed input. List-directed output to a stream file is "readable": character constants are delimited by apostrophes, are preceded and followed by value separators, have each apostrophe in the string represented by a pair of apostrophes (without intervening blanks or end of record) in the file, and do not have blank characters inserted at the beginning of a record that contains the continuation of a character constant from a preceding record; character constants in this form are not suitable for printing.

If two or more successive values have identical values, the processor has the option of producing a single value preceded by a repeat count, for output to a "printable" (sequential) file but not to a "readable" (stream) file.

Slashes as separators, and null values, are not produced by list-directed output formatting.

Except for character constants continued from the previous record in a stream file, each output record begins with a blank character (to provide vertical spacing control when the output file is printed).

3.5 OPEN, CLOSE#, AND INQUIRE# STATEMENTS

3.5.1 Definitions [12.10]

At any given time, there is a processor-determined set of files that are said to exist for an executable program. (Internal files always exist.) To create a file means to cause a file to exist that did not previously exist. To delete a file means to terminate the existence of a file. Any input or output statement may reference a file that exists. An INQUIRE#, OPEN, CLOSE#, WRITE, PRINT, or ENDFILE statement may also reference a file that does not exist. If a file does not exist, successful execution of a WRITE, PRINT, or ENDFILE statement creates the file if it is preconnected; othewise successful execution of an OPEN statement creates the file.

An external unit may be in a connected state with respect to a file, in which case the file is said to be connected to the unit and the unit to the file. A unit may be preconnected by the processor, or it may become connected by execution of an OPEN statement. An input or output statement other than an OPEN, CLOSE#, or INQUIRE# statement must reference (an internal file or) a unit that is connected to a file.

A unit must not be connected to more than one file at a time, and a file must not be connected to more than one unit at a time. However#, the connected relationship between a file and a unit may be terminated by execution of a CLOSE# statement, after which the unit may be reconnected to the same file or a different file, and the file (if it is a named file) may be reconnected to the same unit or to a different unit.

A processor may permit named files#. The name of a named file is a character string; the set of allowable names is processor dependent. The name of a named file is established by execution of an OPEN statement. An INQUIRE# statement may reference the name of a named file. If a file is not named, there is (in general) no way to identify it in order to reconnect it, if it has been disconnected by execution of a CLOSE# statement.

3.5.2 OPEN statement [12.10]

An OPEN statement consists of the keyword OPEN followed by a parenthesized list of specifiers, which must include an external unit specifier (of the same form as for a READ, WRITE, or PRINT statement) and may include at most one of each of the following:

ERR = s, where s# is the label of an executable statement in the same program unit;

FILE = file, where file# is a character expression giving the name of a named file#;

STATUS = status, where status# is 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN';

ACCESS = access, where access is 'SEQUENTIAL'#, 'STREAM'#, or 'DIRECT';

FORM = form#, where form is 'FORMATTED' or 'UNFORMATTED';

RECL = recl, where recl is an integer, real#, or double precision# expression# which, when truncated (if necessary) to an integer has a positive value;

MAXREC = maxrec, where maxrec is an integer, real#, or double precision# expression# which, when truncated (if necessary) to an integer has a positive value;

BLANK = blank#, where blank is 'NULL' or 'ZERO'.

Execution of an OPEN statement in a program unit causes connection for purposes of all other program units of the same executable program as well. An OPEN statement must not cause a unit to become connected to a file that is already connected to a different unit. An OPEN statement that causes a unit to become connected or to remain connected to a file that already exists must not change the access, form#, recl, or maxrec specification of the file. (The blank# specification may be changed, however.) An error condition occurs if an attempt is made to execute an OPEN statement with improperly valued specifiers. The meaning of the specifiers is as follows:

If the file# specifier is given, and the unit is not already connected to a file, then it becomes connected to the specified file, which is created if it does not already exist.

If the file# specifier is given, and the unit is already connected to the specified file, then it remains connected to the same file. If the file does not already exist (is preconnected), it is created.

If the file# specifier is given, and the unit is already connected to a different file, then it is disconnected from the previous file and becomes connected to the specified file, which is created if it does not already exist.

If the file# specifier is omitted, and the unit is already connected to a file, then it remains connected to the same file. If the file does not exist (is preconnected), it is created.

If the file# specifier is omitted, and the unit is not already connected to a file, then it becomes connected to a processor-determined file, which is created if it does not already exist.

If the status# specifier is 'OLD' or 'NEW', a file# specifier must be given. If it is 'OLD', the file must already exist. If it is 'NEW', the file must not already exist; the file will be created and its status will be changed to 'OLD'. If the status specifier is 'SCRATCH', a file specifier must not be given. The default status is 'UNKNOWN'.

The default access specification for an existing file is the existing property; the default access specification for a file being created is 'SEQUENTIAL'.

The form#specifier must not be given except for a direct access file. The default for an existing file is the existing property; the default for a file being created is 'UNFORMATTED'.

The recl specifier gives the length of each record in a direct access file. It is the number of character storage units per record for a formatted file, and the number of noncharacter storage units for an unformatted file. The default for an existing file is the existing length. This specifier must not be omitted for a direct access file that is being created.

The maxrec specifier gives the maximum record number. If this specifier is given when the file is created, the file is said to have the maxrec property. If the specifier is not given when the file is created, the file does not have the maxrec property. For an existing file that has the maxrec property, this specifier must be omitted, or if given must have a value that agrees with the existing value. For an existing file that does not have the maxrec property, this specifier must not be given. A file that does not have the maxrec property has a processor-dependent maximum record number.

The blank# specifier determines whether blank characters in numeric formatted input fields in the file are to be treated as zeros or ignored (except that an entirely blank numeric field is always given the value zero). The default value is 'ZERO'. (Note that a 'NULL' specifier from a previous OPEN will not be preserved, if this specifier is omitted on a subsequent OPEN, since the default value is 'ZERO' independent of the previous specification.)

3.5.3 CLOSE statement# [12.10]

A CLOSE statement consists of the keyword CLOSE followed by a parenthesized list of specifiers, which must include an external unit specifier and may include at most one of each of the following:

ERR = s, where s is the label of an executable statement in the same program unit;

STATUS = status, where status is 'KEEP' or 'DELETE'.

Execution of a CLOSE statement causes termination of the connected relationship between the specified external unit and the file to which it is connected. (If the unit is not connected to a file, the CLOSE statement has no effect.) The effect of a CLOSE statement applies to all program units of the executable program. After execution of a CLOSE statement, the unit may be reconnected to the same file or to a different file, and the file (if it is a named file) may be reconnected to the same unit or to a different unit. If a file is not named, there is (in general) no way to identify it in order to reconnect it, if it has been disconnected by execution of a CLOSE statement.

If the status specifier 'KEEP' is given,

then after execution of the CLOSE statement the file continues in its previous state of existence or nonexistence. If the specifier 'DELETE' is given, the file does not exist after execution of the CLOSE statement. If the specifier is omitted, the default specifier is 'KEEP', except that if the file was previously a 'SCRATCH' file the default specifier is 'DELETE'.

At termination of execution of an executable program for reasons other than an error condition, all units that are connected are implicitly closed with the appropriate default status specifier.

3.5.4 INQUIRE statement# [12.10]

An INQUIRE statement consists of the keyword INQUIRE followed by a parenthesized list of specifiers, which must include either an external unit specifier or a file specifier (but not both) and may include at most one of each of the other specifiers in the list below.

FILE = file, where file is the name of a named file;

ERR = s, where s is the label of an executable statement in the same program unit;

EXIST = exist, where exist is a logical variable or array element;

OPENED = opened, where opened is a logical variable or array element;

NUMBER = number, where number is an integer variable or array element;

NAMED = named, where named is a logical variable or array element;

NAME = name, where name is a character variable or array element;

ACCESS = access, where access is a character variable or array element;

FORM = form, where form is a character variable or array element;

RECL = recl, where recl is an integer variable or array element;

MAXREC = maxrec, where maxrec is an integer variable or array element;

NEXTREC = nextrec, where nextrec is an integer variable or array element.

If a file specifier is given, the INQUIRE statement relates to the specified named file. If no file having the specified name exists, the data referenced by named, name, access, form, recl, maxrec, and nextrec become undefined.

If a unit specifier is given and the specified unit is connected to a file, the inquiry relates to that file. If the specified unit is not connected to a file, the data referenced by number, named, name, access, form, recl, maxrec, and nextrec become undefined. [Do they become undefined if the unit is (pre-) connected to a file that does not exist?]

If the file is not a direct access file, the data referenced by form, recl, maxrec, and nextrec become undefined.

An INQUIRE statement may be given for a file or unit, regardless of whether it is connected. The data referenced by the exists and opened specifiers always become defined unless an error condition occurs. If an error condition occurs, the data referenced by all of the specifiers become undefined. Any datum referenced by more than one specifier in an INQUIRE statement becomes undefined.

The datum referenced by exists becomes true if a file specifier is given and a named file with the specified name exists, or if a unit specifier is given and the unit exists [! What does it mean for a unit to exist? Should the exists inquiry be restricted to the case where a file name is given?] otherwise the datum becomes false.

The datum referenced by opened becomes true if a file specifier is given and the specified file is connected to a unit, or if a unit specifier is given and the specified unit is connected to a file; otherwise the datum becomes false. (This datum becomes undefined if exists becomes undefined, or if exists becomes defined as false.)

The datum referenced by number becomes defined if a file specifier is given and the specified file exists, or if a unit specifier is given and the unit is connected to a file. If a file specifier is given and the specified file is not connected to a unit, the datum is given the value zero; otherwise the datum is given the unit number of the unit connected to the file.

The datum referenced by named becomes true if the file is a named file; othewise the datum becomes false.

The datum referenced by name is assigned the name of the file if the file is a named file; otherwise it becomes undefined. (This name is not necessarily the same as the name given as the file specifier.)

The datum referenced by access is assigned

'SEQUENTIAL', 'STREAM', or 'DIRECT'.

The datum referenced by form is assigned 'FORMATTED' or 'UNFORMATTED', if the file is a direct access file.

The datum referenced by recl is given the record length (measured in character storage units for a formatted file, or in noncharacter storage units for an unformatted file) if the file is a direct access file.

The datum referenced by maxrec is given the maximum record value if the file has the maxrec property; for a direct access file that does not have the maxrec property, the datum is given the value zero.

The datum referenced by nextrec is given a value one greater than the record number of the last record read or written, if any. The datum is given the value 1 if the file is connected but no records have been read or written since the file was connected. The datum becomes undefined if the position of the file is indeterminate due to a previous error condition (or if the file is not a direct access file). [What if the file exists but is not connected?] (Note that the value given to this datum may be as large as maxrec + 1.)

4. MAIN PROGRAM AND SUBPROGRAMS

4.1 MAIN PROGRAM

4.1.1 Main program [14.1, 14.2]

A main program is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA# statement as its first statement; it may have a PROGRAM statement as its first statement.

A main program must not contain a BLOCK DATA#, FUNCTION, SUBROUTINE, ENTRY#, or RETURN statement* and must not contain a PROGRAM statement except as its first statement.

There must be exactly one main program in an executable program. Execution of an executable program begins with the execution of the first executable statement of the main program. A main program must not be referenced as a (subroutine or external function) procedure.

A PROGRAM statement consists of the keyword PROGRAM followed by a symbolic name. This name must not be used as the name of any other program unit, or of a common block, in the same executable program.

---

* Note: SAVE statements are also prohibited in a main program or BLOCK DATA subprogram.

4.2 SUBPROGRAMS

4.2.1 Subroutine subprogram [15.1, 15.6]

A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement. A SUBROUTINE statement consists of the keyword SUBROUTINE, followed by a symbolic name which is the subroutine subprogram name, optionally followed by a parenthesized list of dummy arguments. (If there are no arguments, the enclosing parentheses are omitted.)

Each dummy argument (if any) is the symbolic name of a variable, an array, or a procedure, or is an asterisk# indicating an alternate return#.

A subroutine subprogram may contain any statements except a BLOCK DATA#, FUNCTION, or PROGRAM statement or another SUBROUTINE statement.

A subroutine subprogram must not reference itself, either directly or indirectly.

4.2.2 Function subprogram [15.5]

A function subprogram is a program unit that has a FUNCTION statement as its first statement. A function statement consists of an optional type specification, followed by the keyword FUNCTION, followed by a symbolic name which is the function subprogram name, optionally followed by a parenthesized list of dummy arguments. (If there are no dummy arguments, the enclosing parentheses are optional.)

A type specifier consists of the keyword INTEGER, REAL, DOUBLE PRECISION#, COMPLEX#, or LOGICAL, or of the keyword CHARACTER followed by an unsigned, nonzero integer constant, an integer constant expression# enclosed in parentheses, an asterisk# in parentheses, or the name of an integer variable# that is in a common block or is in every dummy argument list in the function subprogram. If no type specifier appears in a FUNCTION statement, the function subprogram name may appear in a type statement. If the name is of character type, the length specifier for the name must have one of the forms permitted for a length specifier in a FUNCTION statement.

Each dummy argument (if any) is the name of a variable, an array, or a procedure.

The function subprogram name must not appear as a dummy argument name in the FUNCTION statement nor in any ENTRY# statement, and it must not appear in any other nonexecutable statement except a type-statement, in the subprogram.

The function subprogram name must be associated with the name of a local variable of the

same type. This local variable must have the same name as the function subprogram or an entry# in the subprogram. This variable must become defined during execution of the subprogram; its value when a RETURN or END statement is executed is the value returned to the referencing program. If this variable is a character variable of non-constant length#, it must not appear as an operand of a concatenation# operator except in a character assignment statement.

A function subprogram may contain any statement except a BLOCK DATA#, SUBROUTINE, or PROGRAM statement or another FUNCTION statement.

A function subprogram must not reference itself, either directly or indirectly.

4.2.3 Subprogram entry# [15.7]

An ENTRY# statement may cause execution of a subprogram to begin with a statement other than the first executable statement. A subprogram may have zero or more ENTRY statements. An ENTRY statement is nonexecutable, and may appear anywhere within a subroutine or function subprogram except within the range of a DO-loop.

An ENTRY statement consists of the keyword ENTRY, followed by a symbolic name which is the subprogram entry name, optionally followed by a parenthesized list of dummy arguments. (If there are no arguments, the parentheses are omitted in subroutines, optional in functions.)

The dummy arguments (if any) are of the same form as for the subprogram containing the ENTRY statement. The dummy argument list of an ENTRY statement need not agree with that of the FUNCTION or SUBROUTINE statement nor with that of any other ENTRY statement in the subprogram.

A name that appears as a dummy argument in an ENTRY statement must not appear in the same program unit in an executable statement, or in a statement function statement except as a dummy argument of the statement function, if such appearance would precede its first appearance in the program unit in a FUNCTION, SUBROUTINE, or ENTRY statement.

Function subprogram entry#. Each entry name in a function subprogram must be of character type, if and only if the function subprogram name is of character type. All such names of character type must be of the same length.

The entry name must not appear as a dummy argument name in the FUNCTION statement nor in any ENTRY statement, and must not appear in any nonexecutable statement except a type-statement in the subprogram. If the name appears in a character type-statement, the length specifier must have one of the forms permitted in a FUNCTION statement.

The name of a local variable that is the same as the entry name must not appear in any statement except a type-statement, if such appearance would precede the ENTRY statement.

4.2.4 Dummy arguments [15.9]

A dummy argument is the symbolic name of a variable, an array, or a procedure, or in a subroutine it may be an asterisk# indicating an alternate return#.

A dummy argument that is a symbolic name must not appear in an EQUIVALENCE, DATA, PARAMETER#, SAVE, or INTRINSIC statement, nor in a COMMON statement except as a common block name.

A dummy argument may be used as an actual argument in a procedure reference in the subprogram.

A dummy argument name of integer type may appear in an adjustable array declarator or in an adjustable length specifier#.

A character dummy argument of non-constant length# must not appear as an operand of a concatenation# operator except in a character assignment statement.

The array declarator for a dummy argument that is an array name may be an adjustable array declarator; i.e., one or more of its dimension bound expressions may contain integer variable names.

The length specifier for a dummy argument of character type may consist of an asterisk# in parentheses, or it may be an adustable length specifier#. In the former case, the character entity assumes the length of the actual argument. An adjustable length specifier° is an expression that contains one or more integer variable names.

The name of each variable that appears in a dimension bound expression# of an adjustable array declarator, or in an adjustable length specifier#, must appear in the subprogram either in a common block or in every dummy argument list that contains the array name. The actual dimension bound values or length values# are established at the time of the subprogram reference, and do not change (even though the variables involved may be redefined or become undefined) during execution of the subprogram.

If an intrinsic function name appears in a dummy argument list, the name must not be used as an intrinsic function name in the subprogram.

4.3 BLOCK DATA SUBPROGRAM#

4.3.1 BLOCK DATA Subprogram# [16.1 - 16.2]

A block data subprogram provides initial values for variables and array elements in named common blocks.

A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. There may be more than one block data subprogram in an executable program, but an executable program must not include more than one unnamed block data subprogram.

Entities not in a named common block must not be initialized in a block data subprogram. More than one named common block may have entities initialized in a single block data subprogram. A named common block must not be specified in more than one block data subprogram in an executable program. A named common block must have a storage sequence of the same length in a block data subprogram as in all other program units in which it appears.

A BLOCK DATA statement consists of the keywords BLOCK DATA, optionally followed by a symbolic name. A BLOCK DATA statement must not appear except as the first statement of a block data subprogram.

A block data subprogram must not contain any other statements except IMPLICIT, PARAMETER#, DIMENSION, COMMON, EQUIVALENCE, DATA, END, and type-statements. A block data subprogram must not contain any executable statements.

4.4 PROCEDURES

4.4.1 Statement functions [15.4]

A statement function is a procedure specified by a single statement that is similar in form to an assignment statement. A statement function statement is a non-executable statement. It must follow all specification statements and precede all executable statements in the program unit.

A statement function statement consists of a symbolic name (which is the statement function name) followed by a parenthesized dummy argument list, followed by an equals sign (assignment operator), followed by an expression.

The dummy argument list contains zero or more dummy arguments (separated by commas); each dummy argument is a variable name, and the same name must not appear more than once in the list. The length specification of a dummy argument of character type must be an integer constant expression# or an asterisk#.

Both the expression and the statement function procedure name must be of integer, real, double precision#, or complex type#; or both must be of logical type. A statement function procedure must not be of character type.

Variables referenced in the expression may be dummy arguments of the statement function, or variables local to the program unit. The expression must not contain a reference to a statement function unless that statement function appears in a preceding statement function statement in the program unit.

The expression must not include a reference to an external function having side effects that alter the value of a dummy argument of the statement function.

A statement function must not be referenced except in the program unit that contains the statement function statement. The actual arguments in a statement function reference must agree in number and type with the corresponding dummy arguments, and each may be any expression except a character expression that involves concatenation of variable length# operands.

4.4.2 Subroutine procedure reference [15.6]

A subroutine is referenced by a CALL statement, which consists of the keyword CALL followed by the symbolic name of a subroutine procedure or subroutine entry#, optionally followed by a parenthesized list of actual arguments. The actual argument list consists of zero or more actual arguments (separated by commas); each actual argument is an expression (except a character expression# involving concatenation# of variable length# operands), the name of an array or of a procedure (except a function procedure name of character type# or a statement function name), or an alternate return specifier# (consisting of an asterisk followed by the label of an executable statement in the same program unit).

A subroutine procedure reference in a program unit must not cause a direct or indirect reference to that program unit.

4.4.3 Function procedure reference [15.2, 15.3]

A function procedure reference appears in an expression in a program unit. It consists of an external function name, an intrinsic function name, or a statement function name, followed by a parenthesized actual argument list.

The actual argument list contains zero or more actual parameters (separated by commas);

each actual argument is an expression or the name of an array or procedure (except a function procedure name of character type# or a statement function name). Execution of a function reference causes evaluation of those actual arguments that are expressions.

Intrinsic functions. Intrinsic functions are predefined functions supplied by the processor. The intrinsic functions are listed in the table on pages 32 and 33.

An actual argument of an intrinsic function is an expression. (Array names and procedure names are not permitted.)

An intrinsic function may be referenced as a procedure by a specific name or by a generic# name. The type of a function procedure reference using a generic# intrinsic function name depends upon the types of the arguments. If there are two or more actual arguments, they must all be of the same type.

If a specific intrinsic function name appears in a non-conflicting type statement, and the name is not used as an array name, statement function name, or dummy argument name in the same program unit, then the name remains available for use as an intrinsic function name. If a specific intrinsic function name appears in a conflicting type statement, the name must not be used as an intrinsic function name in that program unit.

The names MAX, MIN, LOG, and LOG10 are generic# intrinsic function names but are not specific intrinsic function names; therefore these names are not intrinsic function names when they appear in a type-statement.

If a specific or generic# intrinsic function name appears in the dummy argument list of a subprogram, the name must not be used as an intrinsic function name in the program unit.

Notes for the table of intrinsic functions.

(1) For a of type integer, int (a) = a. For a of type real or double precision#, there are two cases: if $|a| < 1$, int (a) = 0; if $|a| \geq 1$, int (a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For a of type complex#, int (a) is the value obtained by applying the above rule to the real part of a. For a of type real, IFIX (a) is the same as INT (a).

(2) For a of type real, REAL (a) = a. For a of type integer or double precision#, REAL (a) is as much precision of the significant part of a as a real datum can contain. For a of type complex#, REAL (a) is the real part of a.

(3) For a of type double precision#, DBLE (a) = a. For a of type integer or real, DBLE (a) is as much precision of the significant part of a as a double precision datum can contain. For a of type complex#, DBLE (a) is as much precision of the real part of a as a double precision datum can contain.

(4) CMPLX# is a generic# function that may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they must both be of type integer, real, or double precision. For a of type complex, CMPLX (a) = a. For a of type integer, real, or double precision, CMPLX (a) is the complex value whose real part is REAL (a) and whose imaginary part is 0. CMPLX (a_1, a_2) is the complex value whose real part is REAL (a_1) and whose imaginary part is REAL (a_2).

(5) A complex# value is expressed as an ordered pair of reals, (ar, ai), where ar is the real part and ai is the imaginary part.

(6) SIGN (a_1, a_2) is not defined if a_2 is zero.

(7) All angles are expressed in radians.

(8) ATAN is a generic# function that may have one or two arguments. If there is one argument, it may be of real or double precision# type. If there are two arguments, they must both be of real or double precision# type.

(9) The result of a complex# function is the principal value.

(10) All arguments of an intrinsic function must be of the same type.

Restrictions on range of arguments and results. The following restrictions apply to intrinsic functions referenced by their specific names:

(1) The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is zero.

(2) If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, the result is zero, which is neither positive nor negative. The result is undefined when the value of the second argument is zero.

(3) The value of the argument of SQRT or DSQRT# must be non-negative. The result of CSQRT# has non-negative real part, and has non-negative imaginary part when the real part is zero.

Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer int(a) See Note 1	1	INT	-	Integer	Integer
				INT	Real	Integer
				IFIX	Real	Integer
				IDINT	Double	Integer
				-	Complex	Integer
	Conversion to Real See Note 2	1	REAL	FLOAT	Integer	Real
				-	Real	Real
				SNGL	Double	Real
				REAL	Complex	Real
	Conversion to Double See Note 3	1	DBLE	DFLOAT	Integer	Double
				DBLE	Real	Double
				-	Double	Double
				-	Complex	Double
	Conversion to Complex See Note 4	1 or 2	CMPLX	-	Integer	Complex
				CMPLX	Real	Complex
				-	Double	Complex
				-	Complex	Complex
Truncation	int(a) See Note 1	1	AINT	AINT	Real	Real
				DINT	Double	Double
Nearest Whole Number	int(a+.5) if a≥0 int(a-.5) if a<0	1	ANINT	ANINT	Real	Real
				DNINT	Double	Double
Nearest Integer	int(a+.5) if a≥0 int(a-.5) if a<0	1	NINT	NINT	Real	Integer
				IDNINT	Double	Integer
Absolute Value	|a| See Note 5 $(ar^2+ai^2)^{1/2}$	1	ABS	IABS	Integer	Integer
				ABS	Real	Real
				DABS	Double	Double
				CABS	Complex	Real
Remaindering	a1 (modulo a2) a1-int(a1/a2)*a2 See Note 1	2	MOD	MOD	Integer	Integer
				AMOD	Real	Real
				DMOD	Double	Double
Transfer of Sign	|a1| if a2 > 0 -|a1| if a2 < 0 See Note 6	2	SIGN	ISIGN	Integer	Integer
				SIGN	Real	Real
				DSIGN	Double	Double
Positive Difference	a1 - Min(a1,a2)	2	DIM	IDIM	Integer	Integer
				DIM	Real	Real
				DDIM	Double	Double
Double Precision Product	a1*a2	2	-	DPROD	Real	Double
Choosing Largest Value	Max(a1,a2,...)	≥2	MAX	MAX0	Integer	Integer
				AMAX1	Real	Real
				DMAX1	Double	Double
			-	AMAX0	Integer	Real
			-	MAX1	Real	Integer
Choosing Smallest Value	Min(a1,a2,...)	≥2	MIN	MIN0	Integer	Integer
				AMIN1	Real	Real
				DMIN1	Double	Double
			-	AMIN0	Integer	Real
				MIN1	Real	Integer

Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Length	Length of Character Entity	1	-	LEN	Character	Integer
Imaginary Part of Complex Argument	*ai*	1	-	AIMAG	Complex	Real
Conjugate of a Complex Argument	(*ar*,-*ai*) See Note 5	1	-	CONJG	Complex	Complex
Square Root	(*a*)**1/2	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Exponential	e**a	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural Logarithm	log(*a*)	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Logarithm	log10(*a*)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine	sin(*a*)	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Cosine	cos(*a*)	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
Tangent	tan(*a*)	1	TAN	TAN DTAN	Real Double	Real Double
Arcsine	arcsin(*a*)	1	ASIN	ASIN DASIN	Real Double	Real Double
Arccosine	arccos(*a*)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arctangent	arctan(*a*) See Note 8	1	ATAN	ATAN DATAN	Real Double	Real Double
	arctan(a_1/a_2) See Note 8	2	ATAN	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic Sine	sinh(*a*)	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	cosh(*a*)	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	tanh(*a*)	1	TANH	TANH DTANH	Real Double	Real Double

(4) The value of the argument of ALOG, DLOG#, ALOG10, and DLOG10# must be positive. The value of the argument of CLOG# must not be (0.0, 0.0). The range of the imaginary part of the result of CLOG is: $\pi <$ imaginary part $\leq \pi$. The imaginary part is π when the real part of the argument is less than zero and the imaginary part of the argument is zero.

(5) The absolute value of the argument of SIN, DSIN#, COS, DCOS#, TAN, and DTAN# is not restricted to be less than 2π.

(6) The absolute value of the argument of ASIN and DASIN# must be less than or equal to one. The result is between $-\pi/2$*and $\pi/2$*

(7) The absolute value of the argument of ACOS and DACOS# must be less than or equal to one. The range of the result is from 0*to π*

(8) The range of the result for ATAN and DATAN# is from $-\pi/2$*to $\pi/2$* If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, then the result is zero if the second argument is positive and π if the second argument is negative. Both arguments must not be zero. If the value of the first argument is negative, the result is negative. The range of the result is from $-\pi$ to π*

These restrictions also apply to the intrinsic functions when they are referenced by their generic# names.

4.4.4 External functions [15.5]

An external function procedure name is the same as the name of a function subprogram or function subprogram entry# in the same executable program. The type of the corresponding subprogram name or entry# name must be the same as the type of the procedure name, and if these are of character type their lengths must agree. (The length specifier for an external function procedure of character type must not be an adjustable length specifier or an asterisk enclosed in parentheses.)

Each actual argument in an external function procedure reference must be an expression (except a character expression involving concatenation of variable length operands), an array name, an intrinsic function name, or an external function procedure name (except the name of a character function).

An external function procedure reference in a program unit must not directly or indirectly cause a reference to that program unit.

---

* Endpoint so marked is included in range.

4.4.5 Reference to a subprogram entry# [15.7]

An entry name in a subroutine subprogram is referenced as a subroutine procedure name (i.e., by means of a CALL statement). An entry name in a function subprogram is referenced as an external function procedure name (i.e., by the appearance of the name in an expression).

Association of function subprogram entry names#. In an external function subprogram, there is an association between the function subprogram name, all entry names, and all local variable names that are the same as the function name or an entry name. These associated names are not required to be all of the same type (unless one of them is of character type), but at least one associated name of each different type must be a local variable name.

All of these associated entites become undefined at the time the function is referenced (either by the function subprogram name or by an entry name). When an associated local variable becomes defined, all associated entities of the same type become defined and all associated entities of different types become undefined.

An associated local variable of the same type as the name used for the current reference must be in a defined state when a RETURN or END statement in the subprogram is executed.

4.4.6 Actual arguments for an external function or subroutine procedure reference [15.9]

The actual arguments in a procedure reference must agree in number with the dummy arguments of the subprogram.

Corresponding to a dummy argument that is a variable name, the actual argument must be a variable name, an array element name, a substring# name, a constant (including the symbolic name# of a constant), or an expression other than one of the foregoing. If the dummy argument becomes defined during execution of the procedure, the actual argument must be a variable name, an array element name, or a substring# name. The types of the dummy and actual arguments must agree. The length of an actual argument of character type must be greater than or equal to the length of the corresponding dummy argument.

If the actual argument is an expression, it is evaluated at the time the procedure reference occurs. If it is an array element name or a substring# name, the subscript value or delimiting character position# values are determined at the time the procedure reference occurs. These values do not change during execution of the procedure, even though they may involve variables that are redefined or become undefined.

Corresponding to a dummy argument that is an array name, the actual argument must be an array name or an array element name. If the actual argument is the name of an array, its size (number of elements) must equal or exceed that of the dummy array. If it is the name of an array element, then the number of elements between it and the end of the actual array must equal or exceed the size of the dummy array. The types of the dummy and actual arrays must agree. If the dummy array is of character type, then if the actual argument is a character array the total number of characters in the array must equal or exceed the total number of characters in the dummy array; if the actual argument is a character array element, the total number of characters in all the array elements between it and the end of the acutal array must equal or exceed the total number of characters in the dummy array.

Corresponding to a dummy argument name that is a subroutine procedure name, the actual argument must be a subroutine procedure name.

Corresponding to a dummy argument name that is a procedure name other than a subroutine procedure name, the actual argument name must be an external function name of type other than character, or a specific intrinsic function name. Names of intrinsic functions in the "MAX" and "MIN" families must not be used as actual arguments (because the functions in those families have an indefinite number of arguments). Generic intrinsic function names# that are not also specific intrinsic function names are prohibited as actual arguments. Statement function names are prohibited as actual arguments.

If the actual arguments corresponding to two dummy arguments in a procedure, or an entity in a common block in the procedure and an actual argument corresponding to a dummy argument in the same procedure, are the same or associated entites in the referencing program unit, then the dummy arguments must not be redefined nor become undefined during execution of the procedure reference.

4.4.7 RETURN statement [15.8]

A RETURN statement consists of the keyword RETURN, optionally# followed by an integer, real, or double precision# expression#. A RETURN statement must not appear in a main program.

Execution of a RETURN statement or an END statement in a subprogram terminates the subroutine procedure reference or external function reference.

In the execution of an executable program, a subroutine procedure reference or an external function reference must not occur twice without the intervening execution of a RETURN or END statement in the referenced procedure.

Execution of a RETURN or END statement terminates the association between dummy arguments and actual arguments.

Alternate return#. If no expression appears in a RETURN statement of a subroutine subprogram, or if the value of the expression (after truncation to an integer, if necessary) is less than one or greater than the number of asterisks in the dummy argument list of the SUBROUTINE or ENTRY statement referenced by the currently active CALL statement, then execution of the RETURN statement returns control to the statement following the CALL statement. Otherwise, the value of the expression in the RETURN statement (after truncation to an integer, if necessary) designates a particular one of the asterisks in the dummy argument list; control returns to the statement whose label appears in the corresponding alternate return specifier of the actual argument list of the CALL statement.

Definition status. Upon execution of a RETURN or END statement in a subprogram, all local entities become undefined except for the following:

Entities specified by SAVE statements;

Entities in blank common;

Initially defined entities that have not been redefined nor become undefined;

Entities in named common blocks that appear in the subprogram and in at least one other program unit that is currently referencing the subprogram either directly or indirectly.

This work was supported by the United States Energy Research and Development Administration, under contract W-7405-ENG-48.