A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard on April 3, 1978. We report here on a compiler and run-time system for the new extended language. This is believed to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX† systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. An appendix describes the Fortran 77 language.

1 August 1978

†UNIX is a Trademark of Bell Laboratories.

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. for the language, known as Fortran 77, is about to be published. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by SIF, the I/O system by PJW. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler[4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and the Interdata 8/32 UNIX systems. The command to run the compiler is

f 77 *flags file . . .*

f 77 is a general-purpose command for compiling and loading Fortran and Fortran-related files. EFL [6] and Ratfor [7] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The **f 77** and **cc** commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

The following flags are understood:

−S Generate assembler output for each source file, but do not assemble it. Assembler output for a source file **x.f, x.e, x.r,** or **x.c** is put on file **x.s**.

−c Compile but do not load. Output for **x.f, x.e, x.r, x.c,** or **x.s** is put on file **x.o.**
is in use on UNIX†

†UNIX is a Trademark of Bell Laboratories.

−m	Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.
−f	Apply the EFL or Ratfor processor to all relevant files, and leave the output from **x.e** or **x.r** on **x.f.** Do not compile the resulting Fortran program.
−p	Generate code to produce usage profiles.
−o *f*	Put executable module on file *f.* (Default is **a.out**).
−w	Suppress all warning messages.
−w66	Suppress warnings about Fortran 66 features used.
−O	Invoke the C object code optimizer.
−C	Compile code the checks that subscripts are within array bounds.
−onetrip	Compile code that performs every **do** loop at least once. (see Section 2.10).
−U	Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case.
−u	Make the default type of a variable **undefined.** (see Section 2.3).
−I2	On machines which support short integers, make the default integer constants and variables short. (**−I4** is the standard value of this option). (see Section 2.14). All logical quantities will be short.
−E	The remaining characters in the argument are used as an EFL flag argument.
−R	The remaining characters in the argument are used as a Ratfor flag argument.
−F	Ratfor and and EFL source programs are pre-processed into Fortran files, but those files are not compiled or removed.

Other flags, all library names (arguments beginning **−l**), and any names not ending with one of the understood suffixes are passed to the loader.

1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The mathematical functions are computed to at least 63 bit precision. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in the Appendix. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

2.2. Internal Files

The Fortran 77 standard introduces ''internal files'' (memory arrays), but restricts their use to formatted sequential I/O statements. Our I/O system also permits internal files to be used in direct and unformatted reads and writes.

2.3. Implicit Undefined statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m** or **n**, and **real** otherwise. Fortran 77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined.** The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **–u** compiler flag is equivalent to beginning each procedure with this statement.

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic.** These keywords may appear as ''types'' in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

2.6. Source Input Format

The Standard expects input to the compiler to be in 72 column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next sixty-six are the body of the line. (If there are fewer than seventy-two characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored).

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand (''&'') in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **–U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of the flag, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include ′stuff′
```

is replaced by the contents of the file **stuff. include**s may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A **logical, real,** or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0**–**7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0**–**9**, **a**–**f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

```
\nnewline
\ttab
\bbackspace
\fform feed
\0null
\'apostrophe (does not terminate a string)
\"quotation mark (does not terminate a string)
\\\
\xx,  where x is any other character
```

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe (') and the double-quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.
Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

Fortran 77 does not have the old Hollerith (*n* **h**) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.12. One-Trip DO Loops

The Fortran 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **–onetrip** compiler flag causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

(i10, f20.10, i4)

will read the record

–345,.05e–3,12

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer∗2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**.) An expression involving only objects of type **integer∗2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **–I2** flag, all small integer constants will be of type **integer∗2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer∗2** when the **–I2** command flag is in effect). When the **–I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not)** and for accessing the UNIX command arguments (**getarg** and **iargc**).

3. VIOLATIONS OF THE STANDARD

We know only thre ways in which our Fortran system violates the new standard:

3.1. Double Precision Alignment

The Fortran standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c

equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. (Section 6.3.2 in the Appendix.) The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. (People who can make a case for using **tl** should let us know.) A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	C
integer∗2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character∗6 x	char x[6];

(By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory).

4.3. Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function that returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } ∗temp;
 . . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character∗15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
 . . .
```

and could be invoked in C by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3),  nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value). The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
 . . .
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on ‘‘records’’. When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer

containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. (The UNIX operating system implementation attempts to determine the full pathname.) Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist, nor will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end, so a **write** will append to the file and a **read** will result in an end-of-file indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

REFERENCES

1.*Sigplan Notices* **11**, No.3 (1976), as amended in X3J3 internal documents through ''/90.1''.

2.*USA Standard FORTRAN, USAS X3.9-1966*, New York: United States of America Standards Institute, March 7, 1966. Clarified in *Comm. ACM* **12,** 289 (1969) and *Comm. ACM* **14,** 628 (1971).

3.B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall (1978).

4.D. M. Ritchie, private communication.

5.S. C. Johnson, ''A Portable Compiler: Theory and Practice'', Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).

6.S. I. Feldman, ''An Informal Description of EFL'', internal memorandum.

7.B. W. Kernighan, ''RATFOR — A Preprocessor for a Rational Fortran'', *Bell Laboratories Computing Science Technical Report #55*, (January 1977).

8.D. M. Ritchie, private communication.

APPENDIX. Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. At present the only current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute. The following information is from the ''/92'' document. This draft Standard is written in English rather than a meta-language, but it is forbidding and legalistic. No tutorials or textbooks are available yet.

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of ''Hollerith'' (*n* **h**) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick of calling one entry point with a large number of arguments to cause the procedure to ''remember'' the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u - l)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the ''alternate returns'' is described in section 5.2 of the Appendix.

3. Declarations

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer,** other variables are of type **real,** unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real,** those beginning with **w, x, y,** or **z** are assumed **complex,** and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(−5:3, 7, m:n), b(n+1:2∗n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, ∗),  b(∗), c(0:1, −2:∗)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **save**d in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, ‘‘intrinsic functions’’, rather than being divided into ‘‘intrinsic’’ and ‘‘basic external’’ functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
′abc′
′ain′′t′
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, ‘‘ ′ ’’ and ‘‘ " ’’. (See Section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash (‘‘//’’). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
′ab′ // ′cd′
′abcd′
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared

adjustable with a ''∗(∗)'' modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

a(i, j) (m:n)

is the string of $(n - m + 1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leq n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

a**b**c = a ** (b**c)

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameter**s and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in B common..

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a ''Block If''. A Block If begins with a statement of the form

if (. . .) then

and ends with an

end if

statement. Two other new statements may appear in a Block If. There may be several

else if(. . .) then

statements, followed by at most one

else

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif, else,** or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the **elseif** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** must follow all **elseif**s in a Block If. Of course, there may be Block Ifs embedded inside of other

Block If structures). A case construct may be rendered

```
if (s .eq. 'ab') then
 . . .
else if (s .eq. 'cd') then
 . . .
else
 . . .
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as

```
return k
```

If the entry point has *n* alternate return (asterisk) arguments and if $1 \le k \le n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=, err=,** and **iostat=** clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2,'' isn''''t '',i1)') 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn''t ',i1)
```

and the character constant

isn′t

is copied into the output.

6.3.2. Positional Editing Codes

t, tl, tr, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.4 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x=′("hello", :, " there", i4)′
write(6, x) 12
write(6, x)
```

the first **write** statement prints **hello there 12**, while the second only prints **hello**.

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put out optional plus signs, **ss** and **s** codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. Iw.m

There is a new integer output code, **i***w.m.* It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w*.0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w***.1** is the same as **i***w*.

6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. (We do not print it.) There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending. on the size of the number and of *d*.

6.3.9. "A" Format Code

A codes are used for character values. **a***w* use a field width of *w*, while a plain **a** uses the length of the character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.
For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.
Direct access **read** and **write** statements have an extra argument, **rec=,** which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a.**
The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write).** There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

(We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed.)

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 9. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write.**

err= is the same as in **read** or **write.**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status=scratch.**

status= one of **old, new, scratch,** or **unknown.** If this parameter is not given, **unknown** is assumed. If **scratch** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **new** is given, the file will be created if it doesn't exist, or truncated if it does. The meaning of **unknown** is processor dependent; our system treats it as synonymous with **old.**

access= sequential or **direct,** depending on whether the file is to be opened for sequential or direct I/O.

form= formatted or **unformatted.**

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= null or **zero.** This parameter has meaning only for formatted I/O. The default value is **null.** **zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **keep** or **delete.** Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit (''inquire by unit'') or a file (''inquire by file''). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.

formatted= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

unformatted= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.