A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

J. Berkman

University of California
Berkeley, CA 94720

ABSTRACT

The Fortran language has 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. It 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. Appendix A describes the Fortran 77 language extensions.

This is a standard Bell Laboratories document reproduced with minor modifications to the text. The Bell Laboratory's appendix on ''Differences Between Fortran 66 and Fortran 77'' has been changed to Appendix A, and a local appendix has been added. Appendix B contains a list of Fortran 77 references (some from the original Bell document and some added at Berkeley).

Revised September, 1985

† UNIX is a trademark of Bell Laboratories.

1. INTRODUCTION

The Fortran language has been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. 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 S.I.F., the I/O system by P.J.W. 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 is in use on UNIX 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
.F	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

Arguments whose names end with **.f** are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with **.o** substituted for **.f**.

Arguments whose names end with **.F** are also taken to be Fortran 77 source programs; these are first processed by the C preprocessor before being compiled by **f77**.

Arguments whose names end with **.r** or **.e** are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by **f77**.

In the same way, arguments whose names end with **.c** or **.s** are taken to be C or assembly source programs and are compiled or assembled, producing a **.o** file.

The following flags are understood:

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

−d Used in debugging the compiler.

−g Have the compiler produce additional symbol table information for *dbx(1)*. This flag is incompatible with **−O**. See section 1.4 for more details.

−i2 On machines which support short integers, make the default integer constants and variables short (see section 2.14). (**−i4** is the standard value of this option). All logical quantities will be short.

−m Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.

−o *file* Put executable module on file *file* . (Default is **a.out**).

−onetrip or **−1**
Compile code that performs every **do** loop at least once (see section 2.12).

−p Generate code to produce usage profiles.

−pg Generate code in the manner of **−p**, but invoke a run-time recording mechanism that keeps more extensive statistics. See *gprof* (1).

−q Suppress printing of file names and program unit names during compilation.

−r8 Treat all floating point variables, constants, functions and intrinsics as double precision and all complex quantities as double complex. See section 2.17.

−u Make the default type of a variable **undefined** (see section 2.3).

−v Print the version number of the compiler and the name of each pass.

−w Suppress all warning messages.

−w66 Suppress warnings about Fortran 66 features used.

−C Compile code that checks that subscripts are within array bounds. For multi-dimensional arrays, only the equivalent linear subscript is checked.

−D*name=def*

−D*name* Define the *name* to the C preprocessor, as if by '#define'. If no definition is given, the name is defined as "1". (**.F** files only).

−E*str* Use the string *str* as an EFL option in processing **.e** files.

−F Ratfor, EFL, and **.F** source files are pre-processed into **.f** files, and those **.f** files are left on the disk without being compiled.

−I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **−I** options, then in directories on a standard list. (**.F** files only).

−N[**qxscn**]*nnn*
Make static tables in the compiler bigger. The compiler will complain if it overflows its tables and suggest you apply one or more of these flags. These flags have the following meanings:

- **q** Maximum number of equivalenced variables. Default is 150.
- **x** Maximum number of external names (common block names, subroutine and function names). Default is 200.
- **s** Maximum number of statement numbers. Default is 401.
- **c** Maximum depth of nesting for control statements (e.g. DO loops). Default is 20.
- **n** Maximum number of identifiers. Default is 1009.

−O Invoke the object code optimizer. Incompatible with **−g**.

−R*str* Use the string *str* as a Ratfor option in processing **.r** files.

−U Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case except within character string constants.

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

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 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.

1.4. Debugging Aids

A memory image is sometimes written to a file **core** in the current directory upon abnormal termination for errors caught by the **f77** libraries, user calls to **abort**, and certain signals (see **sigvec** (2) in the *UNIX Programmer's Manual*). **Core** is normally created only if the **−g** flag was specified to **f77** during loading.† The source-level debugger *dbx* (1) may be used with the executable and the **core** file to examine the image and determine what went wrong.

In the event that it is necessary to override this default behavior, the user may set the environment variable **f77_dump_flag**. If **f77_dump_flag** is set to a value beginning with **n**, a **core** file is not produced regardless of whether **−g** was specified at compile time, and if the value begins with **y**, dumps are produced even if **−g** was not specified.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in Appendix A. 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 values. The statements

```
z1 = ( 0.1d0, 0.2d0 )
z2 = dcmplx( dx, dy )
```

assign double complex values to **z1** and **z2**. The double precision values which constitute the double complex value may be isolated by using **dreal** or **dble** for the real part and **imag** or **dimag** for the imaginary part. To compute the double complex conjugate of a double complex value, use **conjg** or **dconjg**. The other **double complex** intrinsic functions may be accessed using their generic names or specific names. The generic names are: **abs**, **sqrt**, **exp**, **log**, **sin**, and **cos**. The specific names are the same as the generic names preceded by either **cd** or **z**, e.g. you may code **sqrt**, **zsqrt** or **cdsqrt** to compute the square root of a double complex value.

†Specify **−g** when loading with **cc** or **f77**; specify **−lg** as a library when using **ld** directly.

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 formatted direct reads and writes and list directed sequential read 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. Since Fortran variables are by default **static**, it is often necessary to use the **automatic** storage extension to prevent unexpected results from recursive functions.

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 only one instance of the variable. For variables declared **automatic,** there is a separate instance of the variable for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements. Neither type of variable is guaranteed to retain its value between calls to a subprogram (see the **save** statement in Appendix A).

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 66 are the body of the line. (If there are fewer than 72 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. If **−U** is specified, 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** statements may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A 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:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\′	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
x	*x*, 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.

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** or **–1** compiler flags 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 built-in functions for performing bitwise logical and boolean operations on integer and logical values (**or**, **and**, **xor**, **not**, **lshift**, and **rshift**), and intrinsic functions for **double complex** values (see section 2.1). The **f77** library contains many other functions, such as accessing the UNIX command arguments (**getarg** and **iargc**) and environment (**getenv**). See **intro**(3f) and **bit**(3f) in the *UNIX Programmer's Manual* for more information.

2.16. Namelist I/O

Namelist I/O provides an easy way to input and output information without formats. Although not part of the standard, namelist I/O was part of many Fortran 66 systems and is a common extension to Fortran 77 systems.

Variables and arrays to be used in namelist I/O are declared as part of a namelist in a **namelist** statement, e.g.:

```
character str*12
logical flags(20)
complex c(2)
real arr1(2,3), arr2(0:3,4)
namelist /basic/  arr1, arr2, key, str, c /flglst/ key, flags
```

This defines two namelists: list **basic** consists of variables **key** and **str** and arrays **arr1**, **arr2**, and **c**; list **flglst** consists of variable **key** and array **flags**. A namelist can include variables and arrays of any type, and a variable or array may be in several different namelists. However dummy arguments and array elements may not be in a namelist. A namelist name may be used in external sequential **read**, **write** and **print** statements wherever a format could be used.

In a namelist **read**, column one of each data record is ignored. The data begins with an ampersand in column 2 followed by the namelist name and a blank. Then there is a sequence of value assignments separated by commas and finally an ‘‘&end’’. A simple example of input data corresponding to namelist **basic** is:

```
&basic key=5, str='hi there' &end
```

For compatibility with other systems, dollar signs may be used instead of the ampersands:

```
$basic key=5, str='hi there' $end
```

A value assignment in the data record must be one of three forms. The simplest is a variable name followed by an equal sign followed by a data value which is assigned to that variable, e.g. ‘‘key=5’’. The second form consists of an array name followed by ‘‘=’’ followed by one or more values to be assigned to the array, e.g.:

```
c=(1.1,–2.9),(–1.8e+10,14.0e–3)
```

assigns values to c(1) and c(2) in the complex array c.

As in other **read** statements, values are assigned in the order of the array in memory, i.e. column-major order for two dimensional arrays. Multiple copies of a value may be represented by a repetition count followed by an asterisk followed by the value; e.g. ‘‘3*55.4’’ is the same as ‘‘55.4, 55.4, 55.4’’. It is an error to specify more values than the array can hold; if less are specified, only that number of elements of the array are changed. The third form of a value assignment is a subscripted variable name followed by ‘‘=’’ followed by a value or values, e.g.: ‘‘arr2(0,4)=15.2’’. Only integer constant subscripts may be used. The correct number of subscripts must be used and the subscripts must be legal. This form is the same as the form with an array name except the array is filled starting at the named element.

In all three forms, the variable or array name must be declared in the namelist. The form of the data values is the same as in list directed input except that in namelist I/O, character strings in the data must be enclosed in apostrophes or double quotes, and repetition counts must be followed by data values.

One use of namelist input is to read in a list of options or flags. For example:

```
      logical flags(14)
      namelist /pars/ flags, iters, xlow, xhigh, xinc
      data flags/14*.false./

10    read(5,pars,end=900)
      print pars
      call calc( xlow, xhigh, xinc, flags, iters )
      go to 10
900   continue
      end
```

could be run with the following data (each record begins with a space):

```
 &pars iters=10, xlow=0.0, xhigh=1.0, xinc=0.1 &end
 &pars xinc=0.2,
   flags(2)=2*.true., flags(8)=.true. &end
 &pars xlow=2.0, xhigh=8.0 &end
```

The program reads parameters for the run from the first data set and computes using them. Then it loops and each successive set of namelist input data specifies only those data items which need to be changed. Note the second data set sets the 2^{nd}, 3^{rd}, and 8^{th} elements in the array **flags** to **.true.**.

When a namelist name is used in a **write** or **print** statement, all the values in the namelist are output together with their names. For example the **print** in the program above prints the following:

```
&pars  flags=  f,  f,  f,  f,  f,  f,  f,  f,  f,  f,  f,  f,  f,  f, iters=
  10, xlow=  0., xhigh=   1.00000, xinc=  0.100000
&end
&pars  flags=  f,  t,  t,  f,  f,  f,  f,  t,  t,  f,  f,  f,  f,  f,  f, iters=
  10, xlow=  0., xhigh=   1.00000, xinc=  0.200000
&end
&pars  flags=  f,  t,  t,  f,  f,  f,  f,  f,  t,  f,  f,  f,  f,  f,  f, iters=
  10, xlow=   2.00000, xhigh=   8.00000, xinc=  0.200000
&end
```

Each line begins with a space so that namelist output can be used as input to a namelist **read**. The default is to use ampersands in namelist **print** and **write**. However, dollar signs will be used if the last preceding namelist **read** data set used dollar signs. The character to be used is stored as the first character of the common block **namelistkey**.

2.17. Automatic Precision Increase

The **−r8** flag allows a user to run a program with increased precision without changing any of the program source, i.e. it allows a user to take a program coded in single precision and compile and execute it as if it had been coded in double precision. The option extends the precision of all single precision real and complex constants, variables, external functions, and intrinsic functions. For example, the source:

```
implicit complex(c)
real last
intrinsic sin, csin
data last/0.3/

x = 0.1
y = sqrt(x)+sqrt(last)
c1 = (0.1,0.2)
c2 = sqrt(c1)
x = real(i)
y = aimag(c1)
call fun(sin,csin)
```

is compiled under this flag as if it had been written as:

```
implicit double precision (a-b,d-h,o-z), double complex(c)
double precision last
intrinsic dsin, cdsin
data last/0.3d0/

x = 0.1d0
y = sqrt(x)+sqrt(last)
c1 = (0.1d0,0.2d0)
c2 = sqrt(c1)
x = dreal(i)
y = dimag(c1)
call fun(dsin,cdsin)
```

When the **−r8** flag is invoked, the calls using the generic name **sqrt** will refer to a different specific function since the types of the arguments have changed. This option extends the precision of all single precision **real** and **complex** variables and functions, including those declared **real∗4** and **complex∗8**.

In order to successfully use this flag to increase precision, the entire program including all the subroutines and functions it calls must be recompiled. Programs which use dynamic memory allocation or use equivalence or common statements to associate variables of different types may have to be changed by hand. Similar caveats apply to the sizes of records in unformatted I/O.

2.18. Characters and Integers

A character constant of integer length or less may be assigned to an integer variable. Individual bytes are packed into the integer in the native byte order. The character constant is padded with blanks to the width of the integer during the assignment. Use of this feature is deprecated; it is intended only as a porting aid for extended Fortran 66 programs. Note that the intrinsic **ichar** function behaves as the standard requires, converting only single bytes to integers.

3. VIOLATIONS OF THE STANDARD

We know only a few 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 Appendix A). The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. 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.

3.4. Carriage Control

The Standard leaves as implementation dependent which logical unit(s) are treated as ‘‘printer’’ files. In this implementation there is no printer file and thus by default, no carriage control is recognized on formatted output. This can be changed using **form=′print′** in the **open** statement for a unit, or by using the **fpr**(1) filter for output; see [9].

3.5. Assigned Goto

The optional *list* associated with an assigned **goto** statement is not checked against the actual assigned value during execution.

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 built-in 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. Arrays

The first element of a C array always has subscript zero, while Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order in contiguous storage, C arrays are stored in row-major order. Many mathematical libraries have subroutines which transpose a two dimensional matrix, e.g. **f01crf** in the **NAG** library and **vtran** in the **IMSL** library. These may be used to transpose a two-dimensional array stored in C in row-major order to Fortran column-major order or vice-versa.

4.4. Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function 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( )
```

4.5. 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);
```

4.6. System Interface

To run a Fortran program, the system invokes a small C program which first initializes signal handling, then calls **f_init** to initialize the Fortran I/O library, then calls your Fortran main program, and then calls **f_exit** to close any Fortran files opened.

f_init initializes Fortran units 0, 5, and 6 to standard error, standard input, and standard output respectively. It also calls **setlinebuf** to initiate line buffering of standard error. If you are using Fortran subroutines which may do I/O and you have a C main program, call **f_init** before calling the Fortran subroutines. Otherwise, Fortran units 0, 5, and 6 will be connected to files **fort.0**, **fort.5**, and **fort.6**, and error messages from the **f77** libraries will be written to **fort.0** instead of to standard error. If your C program terminates by calling the C function **exit**, all files are automatically closed. If there are Fortran scratch files to be deleted, first call **f_exit**. **F_init** and **f_exit** do not have any arguments.

The −**d** flag will show what libraries are used in loading Fortran programs.

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. 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. Logical Units and Files

Fortran logical unit numbers may be any integer between 0 and 99. The number of simultaneously open files is currently limited to 48.

Units 5, 6, and 0 are connected before the program begins to standard input, standard output, and standard error respectively.

If an unit is opened explicitly by an **open** statement with a **file=** keyword, then the file name is the name from the **open** statement. Otherwise, the default file name corresponding to unit *n* is **fort.***n*. If there is an environment variable whose name is the same as the tail of the file name after periods are deleted, then the contents of that environment variable are used as the name of the file. See [9] for details.

The default connection for all units 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. The I/O system will position the file at the beginning. Therefore a **write** will destroy any data already in the file, but a **read** will work reasonably. To position a file to its end, use a **read** loop, or the system dependent function **fseek**. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

APPENDIX A: 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. The best current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute, and the ANSI X3.9-1978 document, the official description of the language. The Standard is written in English rather than a meta-language, but it is forbidding and legalistic. A number of tutorials and textbooks are available (see Appendix B).

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 of DO

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 $\max(0, \lfloor (u-l+d)/d \rfloor)$ 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 Appendix A.

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

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

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. Symbolic names for **character** constants may be declared with an implied length ‘‘(*)’’. 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 little known rule of Fortran 66 is that variables in a procedure do not necessarily retain their values between invocations of that procedure. This rule permits overlay and stack implementations for the affected variables. In Fortran 77, three types of variables automatically keep there values: variables in blank common, variables defined in **data** statements and never changed, and variables in named common blocks which have not become undefined. At any instant in the execution of a program, if a named 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 become undefined. 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 an exit from the procedure. 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′
```

Although null (zero-length) character strings are not allowed in the standard Fortran, they may be used with **f77**. 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 character expressions

```
′ab′ // ′cd′
′abcd′
```

are equal.

Dummy arguments of type character may be declared with implied lengths:

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

Such dummy arguments may be used in concatenations in assign statements:

```
s = a // b
```

but not in other contexts. For example:

```
if( a // b .eq. ′abc′ ) key = 1
call sub( a // b )
```

are legal statements if ‘‘a’’ and ‘‘b’’ are dummy arguments declared with explicit lengths, or if they are not arguments. These are illegal if they are declared with implied lengths.

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. Since the two sides of a character assignment must be disjoint, the following are illegal:

```
str = ′ ′ // str
str = str(2:)
```

These are not flagged as errors during compilation or execution, however the result is undefined.

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 $m \le n$. 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 is equivalent to 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 and **format** 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 **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 **else if**, **else,** or **end if** are executed. Otherwise, the next **else if** statement in the group is executed. If none of the **else if** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** block must follow all **else if** blocks 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 integer variables. 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.

A format may be specified as a character constant within the **read** or **write** statement.

```
write(6,′(i2,′′ isn′′′′t ′′,i1)′) 7, 4
```

produces

```
 7 isn′t 4
```

In the example above, the format is the character constant

```
(i2,′ isn′′t ′,i1)
```

and the embedded character constant

```
 isn′t
```

is copied into the output.

The example could have been written more legibly by taking advantage of the two types of quote marks.

```
write(6,′(i2," isn′′t ",i1)′) 7, 4
```

However, the double quote is not standard Fortran 77.

The standard does not allow reading into character constants or Hollerith fields. In order to facilitate running older programs, the Fortran I/O library allows reading into Hollerith fields; however this is a practice to be avoided.

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 3.3 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** or **d**. 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. 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

The **a** code is used for character data. **a***w* uses a field width of *w* , while a plain **a** uses the length of the internal 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 unit is specified by a **print** statement or an asterisk unit:

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

6.5. List-Directed I/O

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. On UNIX, tabs may be used interchangeably with blanks as separators. 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.

The Fortran standard requires data being read into **character** variables by a list-directed read to be enclosed in quotes. In our system, the quotes are optional for strings which do not start with a digit or quote and do not contain separators.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes. According to the standard, they could not be read back using list-directed input. However much of this data could be read back in with list-directed I/O on our system.

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.)** 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 character string 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 two extensions of the standard. The first is 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. In this case a record is a single element of an array of character strings. The second extension is list-directed I/O on internal files.

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= an integer between 0 and 99 inclusive which is the unit to which the file is to be connected (see section 5.3 in the text). 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 file name 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. The meaning of **'unknown'** is processor dependent; our system will create the file if it doesn't exist. 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 must not exist. It will be created for both reading and writing. If **'old'** is given, it is an error for the file not to exist.

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

form= 'formatted' or **'unformatted'.** On UNIX systems, **form='print'** implies **'formatted'** with vertical format control. (See section 3.4 of the text).

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'.** For scratch files the default is **'delete';** 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, name=xx)
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).

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, **′unknown′** if not connected.

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 **′unformatted′** if the file is connected for unformatted I/O, **′formatted′** if the file is connected for formatted I/O, **′print′** for formatted I/O with vertical format control, or **′unknown′** if not connected.

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.

For information on file permissions, ownership, etc., use the Fortran library routines **stat** and **access**.

For further discussion of the UNIX Fortran I/O system see ‘‘Introduction to the f77 I/O Library’’ [9].

APPENDIX B: References and Bibliography

References

1. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*. New York: American National Standards Institute, 1978.
2. *USA Standard FORTRAN, USAS X3.9-1966*. New York: United States of America Standards Institute, 1966. Clarified in *Comm. ACM* 12:289 (1969) and *Comm. ACM* 14:628 (1971).
3. Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* Englewood Cliffs: Prentice-Hall, 1978.
4. Ritchie, D. M. Private communication.
5. Johnson, S. C. ‘‘A Portable Compiler: Theory and Practice,’’ *Proceedings of Fifth ACM Symposium on Principles of Programming Languages*. 1978.
6. Feldman, S. I. ‘‘An Informal Description of EFL,’’ internal memorandum.
7. Kernighan, B. W. ‘‘RATFOR—A Preprocessor for Rational Fortran,’’ *Bell Laboratories Computing Science Technical Report #55*. 1977.
8. Ritchie, D. M. Private communication.
9. Wasley, D. L. ‘‘Introduction to the f77 I/O Library’’, *UNIX Programmer’s Manual, Volume 2c*.

Bibliography

The following books or documents describe aspects of Fortran 77. This list cannot pretend to be complete. Certainly no particular endorsement is implied.

1. Brainerd, Walter S., et al. *Fortran 77 Programming.* Harper Row, 1978.
2. Day, A. C. *Compatible Fortran.* Cambridge University Press, 1979.
3. Dock, V. Thomas. *Structured Fortran IV Programming.* West, 1979.
4. Feldman, S. I. ‘‘The Programming Language EFL,’’ *Bell Laboratories Technical Report*. June 1979.
5. Hume, J. N., and R. C. Holt. *Programming Fortran 77.* Reston, 1979.
6. Katzan, Harry, Jr. *Fortran 77.* Van Nostrand-Reinhold, 1978.
7. Meissner, Loren P., and Organick, Elliott I. *Fortran 77 Featuring Structured Programming*, Addison-Wesley, 1979.
8. Merchant, Michael J. *ABC’s of Fortran Programming.* Wadsworth, 1979.
9. Page, Rex, and Richard Didday. *Fortran 77 for Humans.* West, 1980.
10. Wagener, Jerrold L. *Principles of Fortran 77 Programming.* Wiley, 1980.

Table of Contents