AT&T Bell Laboratories
Murray Hill, NJ 07974

Computing Science Technical Report No. 149

A Fortran-to-C Converter

S. I. Feldman[*]
David M. Gay
Mark W. Maimone[†]
N. L. Schryer

May 16, 1990

[*]Bell Communications Research, Morristown, NJ 07960
[†]Carnegie-Mellon University, Pittsburgh, PA 15213

A Fortran to C Converter

S. I. Feldman

Bellcore
Morristown, NJ 07960

David M. Gay

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Mark W. Maimone

Carnegie-Mellon University
Pittsburgh, PA 15213

N. L. Schryer

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe *f2c*, a program that translates Fortran 77 into C or C++. *F2c* lets one portably mix C and Fortran and makes a large body of well-tested Fortran source code available to C environments.

1. INTRODUCTION

Automatic conversion of Fortran 77 [1] to C [10, 11] is desirable for several reasons. Sometimes it is useful to run a well-tested Fortran program on a machine that has a C compiler but no Fortran compiler. At other times, it is convenient to mix C and Fortran. Some things are impossible to express in Fortran 77 or are harder to express in Fortran than in C (e.g. storage management, some character operations, arrays of functions, heterogeneous data structures, and calls that depend on the operating system), and some programmers simply prefer C to Fortran. There is a large body of well tested Fortran source code for carrying out a wide variety of useful calculations, and it is sometimes desirable to exploit some of this Fortran source in a C environment. Many vendors provide some way of mixing C and Fortran, but the details vary from system to system. Automatic Fortran to C conversion lets one create a *portable* C program that exploits Fortran source code.

A side benefit of automatic Fortran 77 to C conversion is that it allows such tools as *cyntax*(1) and *lint*(1) [4] to provide Fortran 77 programs with some of the consistency and portability checks that the Pfort Verifier [13] provided to Fortran 66 programs. The consistency checks detect errors in calling sequences and are thus a boon to debugging.

This paper describes *f2c*, a Fortran 77 to C converter based on Feldman's original *f*77 compiler [6]. We have used *f2c* to convert various large programs and subroutine libraries to C automatically (i.e., with no manual intervention); these include the PORT3 subroutine library (PORT1 is described in [7, 8]), MINOS [12], and Schryer's floating-point test [14]. The floating-point test is of particular interest, as it relies heavily on correct evaluation of parenthesized expressions and is bit-level self-testing.

As a debugging aid, we sought bit-level compatibility between objects compiled from the C produced by *f2c* and objects produced by our local *f*77 compiler. That is, on the VAX where we developed *f2c*, we sought to make it impossible to tell by running a Fortran program whether some of its modules had been compiled by *f2c* or all had been compiled by *f*77. This meant that *f2c* should follow the same calling conventions as *f*77 [6] and should use *f*77's support libraries, *libF77* and *libI77*.

Although we have tried to make *f2c*'s output reasonably readable, our goal of strict compatibility with *f*77 implies some nasty looking conversions. Input/output statements, in particular, generally get expanded into a series of calls on routines in *libI77*, *f*77's I/O library. Thus the C output of *f2c* would probably be something of a nightmare to maintain as C; it would be much more sensible to maintain the original Fortran, translating it anew each time it changed. Some commercial vendors, e.g., those listed in Appendix A, seek to perform translations yielding C that one might reasonably maintain directly; these translations generally require some manual intervention.

The rest of this paper is organized as follows. Section 2 describes the interlanguage conventions used by *f2c* (and *f*77). §3 summarizes some extensions to Fortran 77 that *f2c* recognizes. Example invocations of *f2c* appear in §4. §5 illustrates various details of *f2c*'s translations, and §6 considers portability issues. §7 discusses the generation and use of *prototypes*, which can be used both by C++ and ANSI C compilers and by *f2c* to check consistency of calling sequences. §8 describes our experience with an experimental *f2c* service provided by *netlib* [5], and §9 considers possible extensions. Appendix A lists some vendors who offer conversion of Fortran to C that one might maintain as C. Finally, Appendix B contains a *man* page telling how to use *f2c*.

2. INTERLANGUAGE CONVENTIONS

Much of the material in this section is taken from [6].

Names

An *f2c* extension inspired by Fortran 8x [3] is that long names are allowed (*f2c* truncates names that are longer than 50 characters), and names may contain underscores. To avoid conflict with the names of library routines and with names that *f2c* generates, Fortran names may have one or two underscores appended. Fortran names are forced to lower case (unless the `-U` option described in Appendix B is in effect); external names, i.e., the names of Fortran procedures and common blocks, have a single underscore appended if they do not contain any underscores and have a pair of underscores appended if they do contain underscores. Thus Fortran subroutines named `ABC`, `A_B_C`, and `A_B_C_` result in C functions named `abc_`, `a_b_c__`, and `a_b_c___`.

Types

The table below shows corresponding Fortran and C declarations; the C declarations use types defined in `f2c.h`, a header file upon which *f2c*'s translations rely. The table also shows the C types defined in the standard version of `f2c.h`.

Fortran	C	standard `f2c.h`
integer∗2 x	shortint x;	short int x;
integer x	integer x;	long int x;
logical x	long int x;	long int x;
real x	real x;	float x;
double precision x	doublereal x;	double x;
complex x	complex x;	struct { float r, i; } x;
double complex x	doublecomplex x;	struct { double r, i; } x;
character∗6 x	char x[6];	char x[6];

By the rules of Fortran, `integer, logical,` and `real` data occupy the same amount of memory, and `double precision` and `complex` occupy twice this amount; *f2c* assumes that the types in the C column above are chosen (in `f2c.h`) so that these assumptions are valid. The translations of the Fortran `equivalence` and `data` statements depend on these assumptions. On some machines, one must modify `f2c.h` to make these assumptions hold. See §6 for examples and further discussion.

Return Values

A function of type `integer`, `logical`, or `double precision` must be declared as a C function that returns the corresponding type. If the `-R` option is in effect (see Appendix B), the same is true of a function of type `real`; otherwise, a `real` function must be declared as a C function that returns `doublereal`; this hack facilitates our VAX regression testing, as it duplicates the behavior of our local Fortran compiler (*f*77). 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

```
void f_(temp, . . .)
complex *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;
ftnlen length;
 . . .
```

and could be invoked in C by

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

Subroutines are invoked as if they were `int`-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 Fortran computed `goto`

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

Argument Lists

All Fortran arguments are passed by address. In addition, for every non-function argument that is of type character, an argument giving the length of the value is passed. (The string lengths are `ftnlen` values, i.e., `long int` quantities passed by value). In summary, the order of arguments is: extra arguments for complex and character functions, an address for each datum or function, and a `ftnlen` for each character argument (other than character-valued functions). 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, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Because Fortran arrays are stored in column-major order, whereas C arrays are stored in row-major order, *f2c* translates multi-dimensional Fortran arrays into one-dimensional C arrays and issues appropriate subscripting expressions.

3. EXTENSIONS TO FORTRAN 77

Since it is derived from *f* 77, *f2c* supports all of the *f* 77 extensions described in [6]. *F2c*'s extensions include the following.

- Type `double complex` (alias `complex*16`) is a double-precision version of `complex`. Specific intrinsic functions for `double complex` have names that start with `z` rather than `c`. An exception to this rule is `dimag`, which returns the imaginary part of a `double complex` value; `imag` is the corresponding generic intrinsic function. The generic intrinsic function `real` is extended so that it returns the real part of a `double complex` value as a `double precision` value; `dble` is the specific intrinsic function that does this job.
- The ''types'' that may appear in an `implicit` statement include `undefined`, which implies that variables whose names begin with the associated letters must be explicitly declared in a type statement. *F2c* also recognizes the Fortran 8x statement

  ```
  implicit none
  ```

  as equivalent to

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

  The command-line option `-u` has the effect of inserting

  ```
  implicit none
  ```

  at the beginning of each Fortran procedure.
- Procedures may call themselves recursively, i.e., may call themselves either directly or indirectly through a chain of other calls.
- The keywords `static` and `automatic` act as ''types'' in type and implicit statements; they specify storage classes. There is exactly one copy of each `static` variable, and such variables retain their values between invocations of the procedure in which they appear. On the other hand, each invocation of a procedure gets new copies of the procedure's `automatic` variables. `Automatic` variables may not appear in `equivalence`, `data`, `namelist`, or `save` statements. The command-line option `-a` changes the default storage class from `static` to `automatic` (for all variables except those that appear in `common`, `data`, `equivalence`, `namelist`, or `save` statements).
- A tab in the first 6 columns signifies that the current line is a free-format line, which may extend beyond column 72. An ampersand `&` in column 1 indicates that the current line is a free-format continuation line. Lines that have neither an ampersand in column 1 nor a tab in the first 6 columns are treated as Fortran 77 fixed-format lines: if shorter than 72 characters, they are padded on the right with blanks until they are 72 characters long; if longer than 72 characters, the characters beyond column 72 are discarded. After taking continuations into account, statements may be up to 1320 characters long; this is the only constraint on the length of free-format lines. (This limit is implied by the Fortran 77 standard, which allows at most 19 continuation lines; $1320 = (1 + 19) \times 66$.)
- Aside from quoted strings, *f2c* ignores case (unless the `-U` option is in effect).
- The statement

```
include stuff
```

is replaced by the contents of the file `stuff`. `Include`s may be nested to a reasonable depth, currently ten. The command-line option `-!I` disables `include`s; this option is used by the *netlib f2c* service described in §8 (for which `include` obviously makes no sense).

- *F*77 allows binary, octal, and hexadecimal constants to appear in `data` statements; *f2c* goes somewhat further, allowing such constants to appear anywhere; they are treated just like a decimal integer constant having the equivalent value. Binary, octal, and hexadecimal constants may assume one of two forms: a letter followed by a quoted string of digits, or a decimal base, followed by a sharp sign #, followed by a string of digits (not quoted). The letter is `b` or `B` for binary constants, `o` or `O` for octal constants, and `x`, `X`, `z`, or `Z` for hexadecimal constants. Thus, for example, `z'a7'`, `16#a7`, `o'247'`, `8#247`, `b'10100111'` and `2#10100111` are all treated just like the integer `167`.
- For compatibility with C, quoted strings may contain the following escapes:

`\0`	null	`\n`	newline
`\\`	\	`\r`	carriage return
`\b`	backspace	`\t`	tab
`\f`	form feed	`\v`	vertical tab
`\'`	apostrophe (does not terminate a string)		
`\"`	quotation mark (does not terminate a string)		
`\`*x*	*x*, where *x* is any other character		

The `-!bs` option tells *f2c* not to recognize these escapes. Quoted strings may be delimited either by double quotes (`"`) or by single quotes (`'`); if a string starts with one kind of quote, the other kind may be embedded in the string without being repeated or quoted by a backslash escape. Where possible, translated strings are null-terminated.

- Hollerith strings are treated as character strings.
- In `equivalence` statements, a multiply-dimensioned array may be given a single subscript, in which case the missing subscripts are taken to be 1 (for backward compatibility with Fortran 66) and a warning message is issued.
- In a formatted read of non-character variables, the I/O library (*libI77*) allows a field to be terminated by a comma.
- Type `real*4` is equivalent to `real`, `integer*4` to `integer`, `real*8` to `double precision`, `complex*8` to `complex`, and, as stated before, `complex*16` to `double complex`.
- The type `integer*2` designates short integers (translated to type `shortint`, which by default is `short int`). Such integers are expected to occupy half a ‘‘unit’’ of storage. The command-line options `-I2` and `-i2` turn type `integer` into `integer*2`; see the *man* page (appendix B) for more details.
- The intrinsic functions `and`, `or`, `xor`, and `not` perform bitwise Boolean operations.
- *LibF77* provides two functions for accessing command-line arguments: `iargc(dummy)` returns the number of command-line arguments (and ignores its argument); `getarg(k,c)` sets the character string `c` to the *k*th command-line argument (or to blanks if *k* is out of range).
- Variable, `common`, and procedure names may be arbitrarily long, but they are truncated after the 50th character. These names may contain underscores (in which case their translations will have a pair of underscores appended).
- MAIN programs may have arguments, which are ignored.
- `Common` variables may be initialized by a `data` statement in any module, not just in a `block data` subprogram.

• The label may be omitted from a `do` loop if the loop is terminated by an `enddo` statement.

• Unnamed Fortran 8x `do while` loops are allowed. Such a loop begins with a statement of the form

`do` [*label*] [`,`] `while(`*logical expression*`)`

and ends either after the statement labelled by *label* or after a matching `enddo`.

• *F2c* recognizes the Fortran 8x synonyms <, <=, ==, >=, >, and <> for the Fortran comparison operators `.LT.`, `.LE.`, `.EQ.`, `.GE.`, `.GT.`, and `.NE.`

• `Namelist` works as in Fortran 8x [3], with a minor restriction on `namelist` input: subscripts must have the form

subscript [: *subscript* [: *stride*]]

For example, the Fortran

```
integer m(8)
real x(10,10)
namelist /xx/ m, x
. . .
read(*,xx)
```

could read

```
&xx x(1,1) = 2, x(1:3,8:10:2) = 1,2,3,4,5,6  m(7:8) = 9,10/
```

but would elicit error messages on the inputs

```
&xx x(:3,8:10:2) = 1,2,3,4,5,6/
&xx x(1:3,8::2)  = 1,2,3,4,5,6/
&xx m(7:) = 9,10/
```

(which inputs would be legal in Fortran 8x). For compatibility with the `namelist` variants supplied by several vendors as Fortran 77 extensions, *f2c*'s version of *libI77* permits `$` to be used instead of `&` and `/` in `namelist` input. Thus the Fortran shown above could read

```
$xx x(1,1) = 2, x(1:3,8:10:2) = 1,2,3,4,5,6  m(7:8) = 9,10$end
```

• Internal list-directed and namelist I/O are allowed.

4. INVOCATION EXAMPLES

To convert the Fortran files `main.f` and `subs.f`, one might use the UNIX® command:

```
f2c main.f subs.f
```

This results in translated files suffixed with `.c`, i.e., the resulting C files are `main.c` and `subs.c`. To translate all the Fortran files in the current directory, compile the resulting C, and create an executable program named `myprog`, one might use the following pair of UNIX commands:

```
f2c *.f
cc -o myprog *.c -lF77 -lI77 -lm
```

The above `-lF77` and `-lI77` options assume that the ''standard'' Fortran support libraries *libF77* and *libI77* are appropriate for use with *f2c*. On some systems this is not the case (as further discussed in §6); if one had installed a combination of the appropriate *libF77* and *libI77* in the appropriate place, then the above example might become

```
f2c *.f
cc -o myprog *.c -lf2c -lm
```

Sometimes it is desirable to use *f2c*'s `-R` option, which tells *f2c* not to force all floating-point operations to be done in double precision. (One might argue that `-R` should be the default, but we find the current arrangement more convenient for testing *f2c*.) With `-R` specified, the previous example becomes

```
f2c -R *.f
cc -o myprog *.c -lf2c -lm
```

Sometimes it is desirable to translate several Fortran source files into a single C file. This is easily done by using *f2c* as a filter:

```
cat *.f | f2c >mystuff.c
```

The `-A` option lets *f2c* use (draft) ANSI C constructs [2], which yields more readable C when `character` variables are initialized. With both `-A` and `-R` specified, the last example becomes

```
cat *.f | f2c -A -R >mystuff.c
```

For use with C++ [15], one would specify `-C++` rather than `-A`; the last example would then become

```
cat *.f | f2c -C++ -R >mystuff.c
```

The `-C++` option gives ANSI-style headers and old-style C formatting of character strings and `float` constants (since some C++ compilers reject the ANSI versions of these constructs).

With ANSI C, one can use *prototypes*, i.e., a special syntax describing the calling sequences of procedures, to help catch errors in argument passing. To make using prototypes convenient, the `-P` option causes *f2c* to create a *file*`.P` of prototypes for the procedures defined in each input *file*`.f` (or *file*`.F`, i.e., the suffix ‘‘`.f`’’ or ‘‘`.F`’’ is replaced by ‘‘`.P`’’). One could concatenate all relevant prototype files into a header file and arrange for the header to be `#include`d with each C file compiled. Since `-P` implies `-A` unless `-C++` is specified, one could convert all the Fortran files in the current directory to ANSI C and get corresponding prototype files by issuing the command

```
f2c -P *.f
```

Several command options may be combined if none but perhaps the last takes an argument; thus to specify `-R` and get C++ prototypes for all the files in the current directory, one could say either

```
f2c -C++ -P -R *.f
```

or

```
f2c -C++PR *.f
```

or

```
f2c -RPC++ *.f
```

— options can come in any order.

For numeric variables initialized by character data, the `-W` option specifies the (machine-dependent!) number of characters per word and is further discussed in §6. This option takes a numeric argument, as in `-W8`; such an option must be listed either separately or at the end of a string of other options, as in

```
f2c -C++RPW8 *.f
```

5. TRANSLATION DETAILS

F2c is based on the ancient *f*77 Fortran compiler of [6]. That compiler produced a C parse-tree, which it converted into input for the second pass of the portable C compiler (PCC) [9]. The compiler has been used for many years and is the direct ancestor of many current Fortran compilers. Thus, it provided us with a solid base of Fortran knowledge and a nearly complete C representation. The converter *f2c* is a copy of the *f*77 Fortran compiler which has been altered to print out a C representation of the program being converted. The program *f2c* is a *horror*, based on ancient code and hacked unmercifully. Users are only supposed to look at its C output, not at its appalling inner workings.

Here are some examples that illustrate *f2c*’s translations. For starters, it is helpful to see a short but complete example: *f2c* turns the Fortran inner product routine

```
      FUNCTION DOT(N,X,Y)
      INTEGER N
      REAL X(N),Y(N)
      DOT = 0
      DO 10 I = 1, N
  10    DOT = DOT + X(I)*Y(I)
      END
```

into

```
/* dot.f -- translated by f2c (version of 11 May 1990  14:38:39).
   You must link the resulting object file with the libraries:
        -lF77 -lI77 -lm -lc   (in that order)
*/

#include "f2c.h"

doublereal dot_(n, x, y)
integer *n;
real *x, *y;
{
    /* System generated locals */
    integer i_1;
    real ret_val;

    /* Local variables */
    static integer i;

    /* Parameter adjustments */
    --y;
    --x;

    /* Function Body */
    ret_val = (float)0.;
    i_1 = *n;
    for (i = 1; i <= i_1; ++i) {
/* L10: */
        ret_val += x[i] * y[i];
    }
    return ret_val;
} /* dot_ */
```

The translated C always starts with a ‘‘translated by f2c’’ comment and a `#include` of `f2c.h`. *F2c* forces the variable and procedure names to lower-case and appends an underscore to the external name `dot` (to avoid possible conflicts with library names). The parameter adjustments ‘‘`--x`’’ and ‘‘`--y`’’ account for the fact that C arrays start at index 0. Unused labels are retained in comments for orienteering purposes. Within a function, Fortran references to the function name are turned into references to the local variable `ret_val`, which holds the value to be returned. Unless the `-R` option is specified, *f2c* converts the return type of `real` function values to `doublereal`. Because using the C ‘‘op=’’ operators leads to greater efficiency on some machines, *f2c* looks for opportunities to use these operators, as in the line ‘‘`ret_val += ...`’’ above.

F2c generally dispenses with superfluous parentheses: ANSI C specifies a clear order of evaluation for floating-point expressions, and *f2c* uses the ANSI C rules to decide when parentheses are required to faithfully translate a parenthesized Fortran expression. Non-ANSI compilers are free to violate parentheses; by default, *f2c* does not attempt to break an expression into several statements to foil pernicious non-ANSI C compilers. Thus, for example, the Fortran

```
      x = a*(b*c)
      y = (a*b)*c
```

becomes

```
x = a * (b * c);
y = a * b * c;
```

The `-kr` and `-krd` options cause *f2c* to use temporary variables to force correct evaluation order with non-ANSI C compilers.

Fortran I/O is complicated; like *f*77, *f2c* converts a Fortran I/O statement into calls on the Fortran I/O library *libI77*. For Fortran `read`s and `write`s, there is generally one call to start the statement, one to end it, and one for each item read or written. Given the Fortran declarations

```
      integer count(10)
      real val(10)
```

the Fortran

```
      read(*,*) count, val
```

is turned into some header lines:

```
static integer c__3 = 3;
static integer c__10 = 10;
static integer c__4 = 4;
. . .
    /* Builtin functions */
    integer s_rsle(), do_lio(), e_rsle();
. . .
    /* Fortran I/O blocks */
    static cilist io__1 = { 0, 5, 0, 0, 0 };
```

and the executable lines

```
s_rsle(&io__1);
do_lio(&c__3, &c__10, (char *)&count[0], (ftnlen)sizeof(integer));
do_lio(&c__4, &c__10, (char *)&val[0], (ftnlen)sizeof(real));
e_rsle();
```

Implicit Fortran do-loops, e.g.

```
      read(*,*) (count(i), val(i), i = 1, 10)
```

get turned into explicit C loops:

```
s_rsle(&io__4);
for (i = 1; i <= 10; ++i) {
    do_lio(&c__3, &c__1, (char *)&count[i - 1], (ftnlen)sizeof(integer));
    do_lio(&c__4, &c__1, (char *)&val[i - 1], (ftnlen)sizeof(real));
}
e_rsle();
```

The Fortran `end=` and `err=` specifiers make the resulting C even less readable, as they require tests to be inserted. For example,

```
      read(*,*,err=10) count, val
 10   continue
```

becomes

```
        i_1 = s_rsle(&io__1);
        if (i_1 != 0) {
            goto L10;
        }
        i_1 = do_lio(&c__3, &c__10, (char *)&count[0], (ftnlen)sizeof(integer));
        if (i_1 != 0) {
            goto L10;
        }
        i_1 = do_lio(&c__4, &c__10, (char *)&val[0], (ftnlen)sizeof(real));
        if (i_1 != 0) {
            goto L10;
        }
        i_1 = e_rsle();
    L10:
        ;
```

A Fortran routine containing n `entry` statements is turned into $n + 2$ C functions, a big one containing the translation of everything but the `entry` statements, and $n + 1$ little ones that invoke the big one. Each little one passes a different integer to the big one to tell it where to begin; the big one starts with a switch that branches to the code for the appropriate entry. For instance, the Fortran

```
      function sine(x)
      data pi/3.14159265358979324/
      sine = sin(x)
      return
      entry cosneg(y)
      cosneg = cos(y+pi)
      return
      end
```

is turned into the big procedure

```
    doublereal sine_0_(n__, x, y)
    int n__;
    real *x, *y;
    {
        /* Initialized data */

        static real pi = (float)3.14159265358979324;

        /* System generated locals */
        real ret_val;

        /* Builtin functions */
        double sin(), cos();

        switch(n__) {
            case 1: goto L_cosneg;
            }

        ret_val = sin(*x);
        return ret_val;

    L_cosneg:
        ret_val = cos(*y + pi);
        return ret_val;
    } /* sine_ */
```

and the little invoking procedures

```
doublereal sine_(x)
real *x;
{
    return sine_0_(0, x, (real *)0);
    }

doublereal cosneg_(y)
real *y;
{
    return sine_0_(1, (real *)0, y);
    }
```

Fortran `common` regions are turned into C `struct`s. For example, the Fortran declarations

```
      common /named/ c, d, r, i, l
      complex c(10)
      double precision d(10)
      real r(10)
      integer i(10)
      logical m(10)

      if (m(i(2))) d(3) = d(4)/d(5)
```

result in

```
struct {
    complex c[10];
    doublereal d[10];
    real r[10];
    integer i[10];
    logical m[10];
} named_;

#define named_1 named_
. . .

    if (named_1.m[named_1.i[1] - 1]) {
        named_1.d[2] = named_1.d[3] / named_1.d[4];
        }
```

Under the `-p` option, the above `if` statement becomes more readable:

```
. . .
#define c (named_1.c)
#define d (named_1.d)
#define r (named_1.r)
#define i (named_1.i)
#define m (named_1.m)
. . .
    if (m[i[1] - 1]) {
        d[2] = d[3] / d[4];
```

If the above `common` block were involved in a `block data` subprogram, e.g.

```
      block data
      common /named/ c, d, r, i, l, m
      complex c(10)
      double precision d(10)
      real r(10)
      integer i(10)
      logical m(10)
      data c(1)/(1.0,0e0)/, d(2)/2d0/, r(3)/3e0/, i(4)/4/,
     *     m(5)/.false./
      end
```

then the `struct` would begin ‘‘`struct named_1_ {`’’, and *f2c* would issue a more elaborate `#define`:

```
#define named_1 (*(struct named_1_ *) &named_)

/* Initialized data */

struct {
    complex e_1;
    doublereal fill_2[10];
    doublereal e_3;
    doublereal fill_4[9];
    real e_5;
    integer fill_6[10];
    integer e_7;
    integer fill_8[11];
    logical e_9;
    integer fill_10[5];
    } named_ = { (float)1., (float)0., {0}, 2., {0}, (float)3., {0}, 4,
            {0},  FALSE_ };
```

In this example, *f2c* relies on C's structure initialization rules to supply zeros to the `fill_`*n* arrays that take up the space for which no `data` values were given. (The logical constants `TRUE_` and `FALSE_` are defined in `f2c.h`.)

Character manipulations of multiple-character strings generally result in function calls. For example, the Fortran

```
      character*(*) function cat(a,b)
      character*(*) a, b
      cat = a // b
      end
```

yields

```
. . .
static integer c__2 = 2;

/* Character */ int cat_(ret_val, ret_val_len, a, b, a_len, b_len)
char *ret_val;
ftnlen ret_val_len;
char *a, *b;
ftnlen a_len;
ftnlen b_len;
{

    /* System generated locals */
    address a_1[2];
    integer i_1[2];

    /* Builtin functions */
    /* Subroutine */ int s_cat();

/* Writing concatenation */
    i_1[0] = a_len, a_1[0] = a;
    i_1[1] = b_len, a_1[1] = b;
    s_cat(ret_val, a_1, i_1, &c__2, ret_val_len);
} /* cat_ */
```

Note how the return-value length (`ret_val_len`) and parameter lengths (`a_len` and `b_len`) are used. Single character operations are generally done in-line. For example, the body of the Fortran

```
        character*1 function lastnb(x,n)
        character*1 x(n)
        lastnb = ' '
        do 10 i = n, 1, -1
           if (x(i) .ne. ' ') then
              lastnb = x(i)
              return
              end if
 10        continue
        end
```

becomes

```
    *ret_val = ' ';
    for (i = *n; i >= 1; --i) {
        if (x[i] != ' ') {
            *ret_val = x[i];
            return ;
        }
/* L10: */
    }
```

F2c uses `struct`s and `#define`s to translate `equivalence`s. For a complicated example showing the interaction of `data` with `common`, `equivalence`, and, for good measure, Hollerith notation, consider the Fortran

```
        common /cmname/ c
        complex c(10)
        double precision d(10)
        real r(10)
        integer i(10)
        logical m(10)
        equivalence (c(1),d(1),r(1),i(1),m(1))
        data c(1)/(1.,0.)/
        data d(2)/2d0/, r(5)/3e0/, i(6)/4/, m(7)/.true./
        call sam(c,d(1),r(2),i(3),m(4),14hsome hollerith,14)
        end
```

The resulting C is

```
. . .
struct cmname_1_ {
    complex c[10];
};

#define cmname_1 (*(struct cmname_1_ *) &cmname_)

/* Initialized data */

struct {
    complex e_1;
    doublereal e_2;
    real e_3;
    integer e_4;
    logical e_5;
    integer fill_6[13];
    } cmname_ = { (float)1., (float)0., 2., (float)3., 4, TRUE_ };


/* Table of constant values */

static integer c__14 = 14;

/* Main program */ MAIN__()
{

    /* Local variables */

#define d ((doublereal *)&cmname_1)
#define i ((integer *)&cmname_1)
#define l ((logical *)&cmname_1)
#define r ((real *)&cmname_1)
    extern /* Subroutine */ int sam_();

    sam_(cmname_1.c, d, &r[1], &i[2], &m[3], "some hollerith", &c__14, 14L);
} /* MAIN__ */

#undef r
#undef l
#undef i
#undef d
```

As this example shows, *f2c* turns a Fortran MAIN program into a C function named `MAIN__`. Why not `main`? Well, *libF77* contains a C main routine that arranges for files to be closed automatically when the Fortran program stops, arranges for an error message to be printed if a floating-point exception occurs, and arranges for the command-line argument accessing functions `iargc` and `getarg` to work properly. This C main routine invokes `MAIN__`.

6. PORTABILITY ISSUES

Three portability issues are relevant to *f2c*: the portability of the support libraries (*libF77* and *libI77*) upon which the translated C programs rely, that of the converter *f2c* itself, and that of the C it produces.

Regarding the first issue, some vendors (e.g., Sun and MIPS) have changed the calling conventions for their *libI77* from the original conventions (those of [6]). Other vendors (e.g., MIPS) have changed the *libF77* calling conventions (e.g., for `complex`-valued functions). Thus, having libraries *libF77* and *libI77* or otherwise having library routines with the names that *f2c* expects is insufficient. When using a machine whose vendor provides but has gratuitously changed *libF77* or *libI77*, one cannot safely mix objects compiled from the C produced by *f2c* with objects compiled by the vendor's Fortran compiler, and one must use the correct libraries with programs translated by *f2c*. In such a case, the recommended procedure is to obtain source for the libraries (e.g. from *netlib* — see §8), combine them into a single library, say `libf2c`, and install the library where it they can be conveniently accessed. On a UNIX system, for example, one might install `libf2c` in `/usr/lib/libf2c.a`; then one could issue the command

```
cc *.c -lf2c -lm
```

to compile and link a program translated by *f2c*.

The converter itself is reasonably portable and has run successfully on Apollo, Cray, IBM, MIPS, SGI, Sun and DEC VAX equipment, all running some version of the UNIX operating system. However, we shall see that the C it produces may not be portable due to subtle storage management issues in Fortran 77. In any case, the C output of *f2c* will run fine, at least if the `-W`*n* option (see Appendix B) is used to set the number of characters per word correctly, and if C `double` values may fall on an odd-word boundary.

The Fortran 77 standard says that `Complex` and `Double Precision` objects occupy two ‘‘units’’ of space while other non-character data types occupy one ‘‘unit.’’ It may be necessary to edit the header file `f2c.h` to make these assumptions hold, if possible. On the Cray, for example, `float` and `double` are the same C types, and Fortran double precision, if available, would correspond to the C type `long double`. In this case, changing the definition of `doublereal` in `f2c.h` from

```
typedef double doublereal;
```

to

```
typedef long double doublereal;
```

would be appropriate. For the Think C compiler on the Macintosh, on the other hand, this line would need to become

```
typedef short double doublereal;
```

If your C compiler predefines symbols that could clash with translated Fortran variable names, then you should also add appropriate `#undef` lines to `f2c.h`. The current default `f2c.h` provides the following `#undef` lines for the following symbols:

```
cray     mc68020  sgi    sun2  u370  u3b5
gcos     mips     sparc  sun3  u3b   unix
mc68010  pdp11    sun    sun4  u3b2  vax
```

As an extension to the Fortran 77 Standard, *f2c* allows noncharacter variables to be initialized with character data. This extension is inherently nonportable, as the number of characters storable per ‘‘unit’’ varies from machine to machine. Since 32 bit machines are the most plentiful, *f2c* assumes 4 characters per Fortran ‘‘unit’’, but this assumption can be overridden by the `-W`*n* command-line option. For example, `-W8` is appropriate for C that is to be run on Cray computers, since Crays store 8 characters per word. An example is helpful here: the Fortran

```
      data i/'abcd'/
      j = i
      end
```

turns into

```
    /* Initialized data */

    static struct {
        char e_1[4];
        } equiv_3 = { {'a', 'b', 'c', 'd'} };

#define i (*(integer *)&equiv_3)

    static integer j;

    j = i;
. . .
#undef i
```

(Some use of `i`, e.g. ‘‘`j = i`’’, is necessary or *f2c* will see that `i` is not used and will not initialize it.) If the target machine were a Cray and the string were `'abcdefgh'` or `"abcdefhg"`, then the Fortran would run fine, but the C produced by *f2c* would only store `"abcd"` in i, 4 being the default number of characters per word. The *f2c* command-line option `-W8` gives the correct initialization for a Cray.

The initialization above is clumsy, using 4 separate characters. Using the option `-A`, for ANSI, produces

```
. . .
        } equiv_3 = { "abcd" };
. . .
```

See Appendix B.

The above examples explain why the Fortran 77 standard excludes Hollerith data statements: the number of characters per word is not specified and hence such code is not portable even in Fortran. (Fortran that conservatively assumes only 1 or 2 characters per word is portable but messy. Note that Fortran 77 forbids the mixing, via `common`, `data`, or `equivalence`, of character and noncharacter types. Like many Fortran compilers, *f2c* permits such nonportable mixing; initialization of numeric variables with Hollerith data is one example of this mixing.)

Some Fortran 66 programs pass Hollerith strings to `integer` variables. *F2c* treats a Hollerith string as a character string, but this may lead to bus errors on some systems if the character string winds up being improperly aligned. The `-h` option instructs *f2c* to try to give character variables and constants the same alignment as `integer`s. Under `-h`, for example, the Fortran

```
      call foo("a string")
      call goo(8ha string)
```

is translated to

```
static struct { integer fill; char val[8+1]; char fill2[3]; } c_b1_st = { 0,
        "a string" };
#define c_b1 c_b1_st.val
. . .
    foo_(c_b1, 8L);
    goo_(c_b1, 8L);
. . .
```

Some systems require that C values of type `double` be aligned on a double-word boundary. Fortran `common` and `equivalence` statements may require some C `double` values to be aligned on an odd-word boundary. On systems where double-word alignment is required, C compilers pad structures, if necessary, to arrange for the right alignment. Often such padding has no effect on the validity of *f2c*’s translation, but using `common` or `equivalence`, it is easy to contrive examples in which the translated C works incorrectly. *F2c* issues a warning message when double-word alignment may cause trouble, but, like *f77*, it makes no attempt to circumvent this trouble; the run-time costs of circumvention would be substantial.

Long decimal strings in `data` statements are passed to C unaltered. However, expressions involving long decimal strings are rounded in a machine-dependent manner. On a VAX 8550, the Fortran

```
      x=1.2**10
      end
```

yields the C

```
    static real x;

    x = (float)6.1917364224000008;
```

ANSI C compilers require that all but one instance of any entity with external scope, such as the `struct`s into which *f2c* translates `common`, be declared `extern` and that exactly one declaration should define the entity, i.e., should not be declared `extern`. Most older C compilers have no such restriction. To be compatible with ANSI usage, the *f2c* command-line option `-ec` causes the `struct` corresponding to an uninitialized `common` region to be declared `extern` and makes a `union` of all successive declarations of that `common` region into a defining declaration placed in a file with the name `cname_com.c`, where `cname` is the name of the `common` region. For example, the Fortran

```
      common /cmname/ c
      complex c(10)
      c(1)=cmplx(1.,0.)
      call sam(c)
      end
      subroutine sam(c)
      complex c
      common /cmname/ca
      complex ca(10)
      ca(2) = cmplx(1e0,2e0)
      return
      end
```

when converted by `f2c -ec` produces

```
/* Common Block Declarations */

union {
    struct {
        complex c[10];
    } _1;
    struct {
        complex ca[10];
    } _2;
} cmname_;

#define cmname_1 (cmname_._1)
#define cmname_2 (cmname_._2)

/* Main program */ MAIN__()
{

    extern /* Subroutine */ int sam_();

    cmname_1.c[0].r = (float)1., cmname_1.c[0].i = (float)0.;
    sam_(cmname_1.c);
} /* MAIN__ */

/* Subroutine */ int sam_(c)
complex *c;
{
    cmname_2.ca[1].r = (float)1., cmname_2.ca[1].i = (float)2.;
    return 0;
} /* sam_ */
```

as well as the file `cmname_com.c`:

```
#include "f2c.h"
union {
    struct {
        complex c[10];
    } _1;
    struct {
        complex ca[10];
    } _2;
} cmname_;
```

The files `*_com.c` may be compiled into a library against which one can load to satisfy overly fastidious ANSI C compilers.

The rules of Fortran 77 apparently permit a situation in which *f2c* declares a function to be of type `int`, then defines it to be of another type, as illustrated by the first example in §7. In that example, *f2c* discovers too late that `f` is not a subroutine. With some C compilers, this causes nothing worse than a warning message; with others, it causes the compilation to be aborted. With unforgiving C compilers, one can usually avoid trouble by splitting the Fortran source into one file per procedure, e.g., with the *fsplit*(1) command, and converting each procedure separately. Another solution is to use prototypes, as discussed in §7.

With an ANSI C system that enforced consistent prototype declarations across separate compilations, it would be impossible to translate the main program correctly in the last example just by looking at the main program. Recent C++ compilers do enforce the consistency of prototype declarations across separate compilations, e.g., by encoding calling sequences into the translated names of functions, except for functions that are declared `extern "C"` and compiled separately. *F2c* allows one to use this escape hatch: under `-C++`, *f2c* inserts

```
#ifdef __cplusplus
extern "C" {
#endif
```

at the beginning of its C++ output and places

```
#ifdef __cplusplus
        }
#endif
```

at the end of its C++ output. The `#ifdef __cplusplus` lines are for the benefit of older C++ compilers that do not recognize `extern "C"`.

7. PROTOTYPES

In ANSI C and C++, a *prototype* describes the calling sequence of a function. Prototypes can save debugging time by helping catch errors in calling sequences. The `-P` option instructs *f2c* to emit prototypes for all the functions defined in the C it produces; specifically, *f2c* creates a *file*`.P` of prototypes for each input *file*`.f` or *file*`.F`. One can then arrange for relevant prototype files to be seen by the C compiler. For instance, if *f2c*'s header file `f2c.h` is installed as `/usr/include/f2c.h`, one could issue the UNIX command

```
cat /usr/include/f2c.h *.P >f2c.h
```

to create a local copy of `f2c.h` that has in it all the prototypes in `*.P`. Since the C produced by *f2c* always specifies

```
#include "f2c.h"
```

(rather than `#include <f2c.h>`), the C compiler will look first in the current directory for `f2c.h` and thus will find the local copy that contains the prototypes.

F2c can also read the prototype files it writes; one simply specifies them as arguments to *f2c*. In fact, *f2c* reads all prototype files before any Fortran files; although multiple Fortran files are handled independently, any prototype file arguments apply to all of them. *F2c* has more detailed knowledge of Fortran types than it conveys in the C it puts out; for example, `logical` and `integer` are different Fortran types, but are mapped to the same C type. Moreover, `character`, `complex`, and `double complex` Fortran functions are all translated to `VOID` C functions, and, unless the `-R` option is specified, both `real` and `double precision` Fortran functions are translated to `doublereal` C functions. Because *f2c* denotes all these types differently in its prototype files, it can catch errors that are invisible to an ANSI C (or C++) compiler.

The following table shows the types that *f2c* uses for procedure arguments:

`C_fp`	`complex`
`D_fp`	`doublereal`
`E_fp`	`real` under `-!R` (the default)
`H_fp`	`character`
`I_fp`	`integer` or `integer*4`
`J_fp`	`integer*2`
`K_fp`	`shortlogical` (`logical` under `-i2` or `-I2`)
`L_fp`	`logical`
`R_fp`	`real` under `-R`
`S_fp`	`subroutine`
`U_fp`	untyped `external`
`Z_fp`	`doublecomplex`

These types are defined in `f2c.h`; they appear in prototypes and, under `-A` or `-C++`, in the C that *f2c* writes. Prototypes also use special `void` types to denote the return values of `complex`, `double complex`, and `character` functions:

`C_f`	`complex`
`H_f`	`character`
`Z_f`	`double complex`

F2c also writes special comments in prototype files giving the length of each `common` block; when given prototype files as arguments, *f2c* reads these special comments so it can issue a warning message if its Fortran input specifies a different length for some `common` block.

Sometimes people write otherwise valid Fortran 77 that specifies different lengths for a `common` block. If such Fortran is split into several files and converted to C, the loader could end up giving too little space to the `common` block in question. One can avoid the confusion this could cause by running *f2c* twice, first with `-P!c`, then with the resulting prototypes as additional arguments; the prototypes let *f2c* determine (and convey to all of its output C files) the true length needed for each `common` block.

One complication with prototypes comes from Fortran subprograms that declare a procedure to be `external` but do not explicitly specify a type for it and only pass it as a parameter to another procedure. (If the subprogram also invokes the `external` procedure, then *f2c* can tell whether the procedure is a subroutine or a function; in the latter case, Fortran's implicit typing rules specify a type for the procedure.) If it can do no better, then *f2c* assumes that untyped `external` procedures are subroutines (and hence become `int`-valued functions in C). This can cause the generated C to have multiple and inconsistent declarations for some procedures. For example,

```
external f
call foo(f)
end
function f(x)
double precision f, x
f = x
end
```

results in `MAIN__` declaring

```
extern /* Subroutine */ int f_();
```

and in the subsequent definition of `doublereal f_(x)` in the same C file. Such inconsistencies are grounds for some C compilers to abort compilation.

F2c's type inferences only apply sequentially to the procedures in a file, because *f2c* writes C for each procedure before reading the next one. Thus, as just illustrated, if procedure `xyz` comes after `abc` in a Fortran input file, then *f2c* cannot use information it gains when it sees the definition of `xyz` to deduce types for `external` procedures passed as arguments to `xyz` by `abc`. By using the `-P` option and running *f2c* several times, one can get around this deficiency. For instance, if file `zap.f` contains the Fortran shown above, then the commands

```
f2c -P!c zap.f
f2c -A zap.[fP]
```

result in a file `zap.c` in which `MAIN__` correctly types `f_` and `foo_` as

```
extern doublereal f_();
extern /* Subroutine */ int foo_(D_fp);
```

rather than

```
extern /* Subroutine */ int f_();
extern /* Subroutine */ int foo_(U_fp);
```

The first invocation of *f2c* results in a file `zap.P` containing

```
extern doublereal f_(doublereal *x);
/*:ref: foo_ 10 1 200 */
```

The second invocation of *f2c* is able to type `f_` and `foo_` correctly because of the first line in `zap.P`.

The second line in `zap.P` is a special comment that records the incomplete type information that *f2c* has about `foo_`. *F2c* puts one such special comment in the prototype file for each Fortran procedure that is referenced but not defined in the Fortran file. When it reads prototype files, *f2c* deciphers these comments and uses them to check the consistency of calling sequences. As it learns more about untyped external procedures, *f2c* updates the information it has on them; the `:ref:` comments it writes in a prototype file reflect *f2c*'s latest knowledge.

Ordinarily *f2c* tries to infer the type of an untyped `external` procedure from its use as arguments to procedures of known argument types. For example, if `f.f` contains just

```
external f
call foo(f)
end
```

and if `foo.P` contains

```
extern int foo_(D_fp);
```

then

```
f2c -A f.f foo.P
```

results in the declaration

```
extern doublereal f_();
```

Under unusual circumstances, such type inferences can lead to erroneous error messages or to incorrect typing. Here is an example:

```
subroutine zoo
external f
double precision f
external g
call zap(1,f)
call zap(2,g)
end
subroutine goo
call g
end
```

F2c first infers g to be a double precision function, then discovers that it must be a subroutine and issues a warning message about inconsistent declarations for `g`. This example is legal Fortran 77; `zap` could be defined, for instance, by

```
subroutine zap(n,f)
external f
if (n .le. 1) call zap1(f)
if (n .ge. 2) call zap2(f)
end
```

In such a case one can specify the `-!it` option to instruct *f2c* not to infer the types of otherwise untypable `external` procedures from their appearance as arguments to known procedures. Here is another (somewhat far-fetched) example where `-!it` is useful:

```
subroutine grok(f,g,h)
external f, g, h
logical g
call foo(1,g)
call foo(2,f)
call zit(1,f)
call zit(2,h)
call zot(f(3))
end
```

Without `-!it`, *f2c* first infers `f_` to be a `logical` function, then discovers that Fortran's implicit typing rules require it to be a `real` function. *F2c* issues the warning message ``fixing wrong type inferred for f'', which should serve as a warning that *f2c* may have made some incorrect type inferences in the mean time. Indeed, *f2c* ends up typing `h_` as a `logical` function; with `-!it` specified, *f2c* types `h_` as an `external` procedure unknown type, i.e., a `U_fp`, which to the C compiler appears to be a subroutine. (Even with `-!it` specified, *f2c* issues a warning message about inconsistent calling sequences for `foo`.)

Because *f2c* writes its latest knowledge of types into prototype files, it is easy to write a crude (Bourne) shell script that will glean the maximum possible type information:

```
>f.p
until
        f2c -Pit f.p f.f
        cmp -s f.p f.P
do
        mv f.P f.p
        done
```

In such scripts, use of the `-Ps` option can save an iteration; `-Ps` implies `-P` and instructs *f2c* to issue return code 4 if another iteration might change a declaration or prototype. Thus the following script is more efficient:

```
while :; do
        f2c -Ps f.[fP]
        case $? in 4) ;; *) break;; esac
        done
```

The number of iterations depends on the call graph of the procedures in `f.f` and on their order of appearance in `f.f`. Sorting them into topological order (so that if `abc` calls `def`, then `abc` precedes `def`) and reverse topological order and alternating between the two orders is probably a good heuristic. For example, we were able to completely type the PORT3 subroutine library in two passes by first processing it in reverse topological order, then in forward order. Unfortunately, one can devise situations where arbitrarily many iterations are required. This is slightly annoying, since with appropriate data structures (in an extensively reorganized version of *f2c*), one could do this calculation in linear time.

8. EXPERIENCE WITH *netlib*

With the help of Eric Grosse, we arranged for the *netlib* [5] server `netlib@research.att.com` to provide an experimental Fortran-to-C translation service by electronic mail. By executing the UNIX command

```
(echo execute f2c; cat foo.f) | mail netlib@research.att.com
```

one submits the Fortran in `foo.f` to *netlib*'s *f2c* service; *netlib* replies with the C and diagnostic messages produced by *f2c* from `foo.f`. (The `include` mechanism described in §3 makes no sense in this context, so it is disabled.) To start using this service, one would generally execute

```
echo 'send index from f2c' | mail netlib@research.att.com
```

to check on the current status of the service. Before compiling the returned C, it is necessary to get a copy of `f2c.h`:

```
echo 'send f2c.h from f2c' | mail netlib@research.att.com
```

Most likely it would also be necessary to obtain source for the versions of *libF77* and *libI77* assumed by *f2c*:

```
echo 'send libf77 libi77 from f2c' | mail netlib@research.att.com
```

For testing purposes, we retain the original Fortran submitted to *netlib*'s ‘‘`execute f2c`’’ service. Observing *f2c*'s behavior on over 400,000 lines of submitted Fortran helped us find many obscure bugs and led us to make some of the extensions described in §3. For example, a `block data` subprogram initializing a variable that does not appear in any `common` blocks now elicits a warning message (rather than causing *f2c* to drop core). Another example is that *f2c* now gives the warning message ‘‘`Statement order error: declaration after DATA`’’ and declines to produce any C if a declaration comes after a `data` statement (for reasons discussed in §9); *f2c* formerly gave a more obscure error message and then produced invalid C.

Now that *netlib* offers source for *f2c* itself (as explained in the `index` file mentioned above), we expect to curtail *netlib*'s ''`execute f2c`'' service, perhaps limiting it to employees of AT&T and Bellcore; to learn the current state of affairs, request the current `index` file.

9. POSSIBLE EXTENSIONS

Currently *f2c* simplifies constant expressions. It would be nice if constant expressions were simply passed through, and if Fortran `parameter`s were translated as `#define`s. Unfortunately, several things conspire to make this nearly impossible to do in full generality. Perhaps worst is that `parameter`s may be assigned `complex` or `doublecomplex` expressions that might, for example, involve complex division and exponentiation to a large integer power. `Parameter`s may appear in `data` statements, which may initialize `common` variables and so be moved near the beginning of the C output. Arranging to have the right `#define`s in effect for the data initialization would, in this worst case, be a nightmare. Of course, one could arrange to handle ''easy'' cases with unsimplified constant expressions and `#define`s for parameters.

Prototypes and the argument consistency checks currently ignore alternate return specifiers. Prototypes could be adorned with special comments indicating where alternate return specifiers are supposed to come, or at least telling the number of such specifiers, which is all that really matters. Since alternate return specifiers are rarely used (Fortran 8x calls them ''obsolescent''), we have so far refrained from this exercise.

Fortran 8x allows `data` statements to appear anywhere. It would be nice if *f2c* could do the same, but that would entail major rewriting of *f2c*. Presently `data` values are written to a file as soon as they are seen; among the information in the file is the offset of each value. If an `equivalence` statement could follow the `data` statement, then the offsets would be invalidated.

It would be fairly straightforward to extend *f2c*'s I/O to encompass the new specifiers introduced by Fortran 8x. Unfortunately, that would mean changing *libI77* in ways that would make it incompatible with *f*77.

Of course, it would be nice to translate all of Fortran 8x, but some of the Fortran 8x array manipulations would require new calling conventions and large enough revisions to *f2c* that one might be better off starting from scratch.

With sufficient hacking, *f2c* could be modified to recognize Fortran 8x control structures (`case`, `cycle`, `exit`, and named loops), local arrays of dimensions that depend on arguments and common values, and such types as `logical*1`, `logical*2`, `integer*1` or `byte`. Since our main concern is with making portable Fortran 77 libraries available to the C world, we have so far refrained from these further extensions. Perhaps commercial vendors will wish to provide some of these extensions.

10. REFERENCES

[1] *American National Standard Programming Language FORTRAN,* American National Standards Institute, New York, NY, 1978. ANSI X3.9-1978.

[2] *Draft Proposed American National Standard for Information Systems — Programming Language C,* CBEMA, 1988. Document Number X3J11/88-158.

[3] *American National Standard for Information Systems Programming Language Fortran,* CBEMA, 1989. Draft S8, Version 112.

[4] *UNIX Time Sharing System Programmer's Manual,* AT&T Bell Laboratories, 1990. Tenth Edition, Volume 1.

[5] J. J. Dongarra and E. Grosse, ''Distribution of Mathematical Software by Electronic Mail,'' *Communications of the ACM* **30** #5 (May 1987), pp. 403−407.

[6] S. I. Feldman and P. J. Weinberger, ''A Portable Fortran 77 Compiler,'' in *Unix Programmer's Manual, Volume II,* Holt, Rinehart and Winston (1983).

[7] P. A. Fox, A. D. Hall, and N. L. Schryer, ''Algorithm 528: Framework for a Portable Library,'' *ACM Trans. Math. Software* **4** (June 1978), pp. 177−188.

[8] P. A. Fox, A. D. Hall, and N. L. Schryer, ''The PORT Mathematical Subroutine Library,'' *ACM Trans. Math. Software* **4** (June 1978), pp. 104−126.

[9] S. C. Johnson, ''A Portable Compiler: Theory and Practice,'' pp. 97−104 in *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery (1978).

[10] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.

[11] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1988. Second Edition

[12] B. A. Murtagh and M. A. Saunders, ''MINOS 5.1 User's Guide,'' Technical Report SOL 83-20R (1987), Systems Optimization Laboratory, Stanford University, Stanford, CA.

[13] B. G. Ryder, ''The PFORT Verifier,'' *Software Practice and Experience* **4** (1974), pp. 359−377.

[14] N. L. Schryer, ''A Test of a Computer's Floating-point Arithmetic Unit,'' in *Sources and Development of Mathematical Software*, ed. W. Cowell, Prentice-Hall (1981).

[15] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

Appendix A: Commercial Fortran-to-C Vendors

At the time of this writing, we are aware that the following vendors offer Fortran to C conversion service. Omitted vendors are invited to inform us of their existence, so we may include them in updated versions of this appendix.

Cobalt Blue
2940 Union, Suite C
San Jose, CA 95124
(408) 723−0474

PROMULA Development Corporation
Columbus, OH
(614) 263−5454

Rapitech Systems
Office Center at Montebello
400 Rella Blvd.
Suffern, NY 10901
(914) 368−3000

NAME

f2c – Convert Fortran 77 to C or C++

SYNOPSIS

`f2c` [*option* ...] *file* ...

DESCRIPTION

F2c converts Fortran 77 source code in *files* with names ending in `.f` or `.F` to C (or C++) source files in the current directory, with `.c` substituted for the final `.f` or `.F`. If no Fortran files are named, *f2c* reads Fortran from standard input and writes C on standard output. *File* names that end with `.p` or `.P` are taken to be prototype files, as produced by option `-P`, and are read first.

The following options have the same meaning as in *f77*(1).

`-C` Compile code to check that subscripts are within declared array bounds.

`-I2` Render INTEGER and LOGICAL as short, INTEGER∗4 as long int. Assume the default *libF77* and *libI77*: allow only INTEGER∗4 (and no LOGICAL) variables in INQUIREs. Option `-I4` confirms the default rendering of INTEGER as long int.

`-onetrip`
Compile DO loops that are performed at least once if reached. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)

`-U` Honor the case of variable and external names. Fortran keywords must be in *lower* case.

`-u` Make the default type of a variable ‘undefined’ rather than using the default Fortran rules.

`-w` Suppress all warning messages. If the option is `-w66`, only Fortran 66 compatibility warnings are suppressed.

The following options are peculiar to *f2c*.

`-A` Produce ANSI C. Default is old-style C.

`-a` Make local variables automatic rather than static unless they appear in a DATA, EQUIVALENCE, NAMELIST, or SAVE statement.

`-C++` Output C++ code.

`-c` Include original Fortran source as comments.

`-E` Declare uninitialized COMMON to be `Extern` (overridably defined in `f2c.h` as `extern).`

`-ec` Place uninitialized COMMON blocks in separate files: `COMMON /ABC/` appears in file `abc_com.c`. Option `-e1c` bundles the separate files into the output file, with comments that give an unbundling *sed*(1) script.

`-ext` Complain about *f77*(1) extensions.

`-g` Include original Fortran line numbers as comments.

`-h` Try to align character strings on word (or, if the option is `-hd`, on double-word) boundaries.

`-i2` Similar to `-I2`, but assume a modified *libF77* and *libI77* (compiled with `-Df2c_i2`), so INTEGER and LOGICAL variables may be assigned by INQUIRE.

`-kr` Use temporary values to enforce Fortran expression evaluation where K&R (first edition) parenthesization rules allow rearrangement. If the option is `-krd`, use double precision temporaries even for single-precision operands.

`-P` Write a *file*`.P` of ANSI (or C++) prototypes for procedures defined in each input *file*`.f` or *file*`.F`. When reading Fortran from standard input, write prototypes at the beginning of standard output. Implies `-A` unless option `-C++` is present. Option `-Ps` implies `-P` , and gives exit status 4 if rerunning *f2c* may change prototypes or declarations.

`-p` Supply preprocessor definitions to make common-block members look like local variables.

`-R` Do not promote REAL functions and operations to DOUBLE PRECISION. Option `-!R` confirms the default, which imitates *f77*.

`-r8` Promote REAL to DOUBLE PRECISION, COMPLEX to DOUBLE COMPLEX.

`-T`*dir* Put temporary files in directory *dir*.

`-w8` Suppress warnings when COMMON or EQUIVALENCE forces odd-word alignment of doubles.

`-W`*n* Assume *n* characters/word (default 4) when initializing numeric variables with character data.

`-z` Do not implicitly recognize DOUBLE COMPLEX.

`-!bs` Do not recognize *b*ack*s*lash escapes (\", \', \0, \\, \b, \f, \n, \r, \t, \v) in characer strings.

`-!c` Inhibit C output, but produce `-P` output.

`-!I` Reject `include` statements.

`-!it` Don't infer types of untyped EXTERNAL procedures from use as parameters to previously defined or prototyped procedures.

`-!P` Do not attempt to infer ANSI or C++ prototypes from usage.

The resulting C invokes the support routines of *f77*; object code should be loaded by *f77* or with *ld*(1) or *cc*(1) options `-lF77 -lI77 -lm`. Calling conventions are those of *f77*: see the reference below.

FILES

file`.[fF]`
  input file

`*.c` output file

`/usr/include/f2c.h`
  header file

`/usr/lib/libF77.a`
  intrinsic function library

`/usr/lib/libI77.a`
  Fortran I/O library

`/lib/libc.a`
  C library, see section 3

SEE ALSO

S. I. Feldman and P. J. Weinberger, 'A Portable Fortran 77 Compiler', *UNIX Time Sharing System Programmer's Manual*, Tenth Edition, Volume 2, AT&T Bell Laboratories, 1990.

DIAGNOSTICS

The diagnostics produced by *f2c* are intended to be self-explanatory.

BUGS

Floating-point constant expressions are simplified in the floating-point arithmetic of the machine running *f2c*, so they are typically accurate to at most 16 or 17 decimal places.
Untypable EXTERNAL functions are declared `int`.